“We’re not in Kansas anymore.”

Dorothy

EDA031 C++ Programming – Course and slides

“If I have seen further it is by standing on the sholders of Giants.”
Isaac Newton

Course and course material developed and updated continuously for 20 years by Per Holm – now retired.

History of C++

C++
: C + Simula (object orientation) + multiple inheritance + overloading + templates + exceptions. Invented by Bjarne Stroustrup.

1980
: C with classes

1983
: C++ (C with classes + virtual functions + overloading + more)

Later
: New versions containing multiple inheritance, templates, exceptions

1998
: ISO standard, defines standard library

2011
: New ISO standard, C++11

2014
: Cleanup of C++11

C++11, the New Standard C++11

Stroustrup: "C++11 feels like a new language." C++11 has many features that makes the language easier to learn and to use.

Work on a new C++ standard started early. Initially the new standard was called C++0X, and it was hoped that "X would be a decimal digit." That didn't happen, but the standard was finally accepted in 2011.

These slides (and Lippman's book) describe the new standard. Additions to the base language and to the library are marked `C++11`.

Hello, world . . .

```
#include <iostream>  // input-output library

int main() {
    std::cout << "Hello, world" << std::endl;
}
```

- `main()` is a "free" function, not a method in a class
- Can have arguments similar to `String[] args` in Java
- The return type is `int`, returns the status code. `0` success, `!= 0` failure (`return 0` may be omitted in `main`, but not in other functions)
- `std::` is used to fetch names from the `std` "namespace" (similar to the Java package `java.lang`)

Compile, Link and Run

The Hello world program is in a file *hello.cc*.

```
g++ -c hello.cc       // compile, hello.cc -> hello.o
g++ -o hello hello.o // link, hello.o + library -> hello
./hello               // execute
```

- Usually, there are several more options on the command lines
- Usually, an executable consists of several `.o` files
- g++ is the GNU C++ compiler
- `clang++` is another compiler, from the LLVM project

Parameters on the Command Line

You can access parameters given on the command line if you define the main function like this:

```
int main(int argc, char* argv[]) { ... }
```

argc is the number of parameters, including the program name. argv is an array of C-style strings that contains the parameters (more about strings later). Example:

```
./myprog -a myfile.txt
    argc = 3
    argv[0] = "./myprog"
    argv[1] = "-a"
    argv[2] = "myfile.txt"
    argv[3] = 0             // end mark
```

Reading and Writing

```
#include <iostream>

int main() {
    int sum = 0;
    int value;
    while (std::cin >> value) {
        sum += value;
    }
    std::cout << "The sum is: " << sum << std::endl;
}
```

- `cin` and `cout` are input and output streams (stdin and stdout).
- `>>` is the input operator, reads whitespace-separated tokens. Returns `false` on end-of-file.
- `<<` is the output operator.

Reading and Writing Files

You can create stream objects in order to read from or write to files (`ifstream` for reading, `ofstream` for writing). Read from a file whose name is given as the first argument on the command line:

```
#include <fstream>  // file streams
#include <iostream>

int main(int argc, char* argv[]) {
    std::ifstream input(argv[1]);
    if (!input) { // "if the file couldn't be opened"
        cerr << "Could not open: " << argv[1] << endl;
        exit(1);
    }
    // ... read with input >> ...
    input.close();
}
```

In C++11, the filename can also be a `string` object. C++11

A Class Definition

```
class Point {
public:
    Point(double ix, double iy); // constructor
    double getX() const;         // const: does not modify
    double getY() const;         // the state of the object
    void move(double dx, double dy); // does modify, not const
private:
    double x;                    // attributes
    double y;
};                               // note semicolon
```

- Methods are called member functions in C++, attributes are called member variables
- Member functions are only declared here, the definitions are in a separate file
- `public` and `private` sections, arbitrary order, `private` is default

Definition of Member Functions

```
Point::Point(double ix, double iy) {
    x = ix; // this is ok but there are better ways to
    y = iy; // initialize member variables, see next slide
}

double Point::getX() const {
    return x;
}

double Point::getY() const {
    return y;
}

void Point::move(double dx, double dy) {
    x += dx;
    y += dy;
}
```

- `Point::` indicates that the functions belong to class Point

Initialization of Member Variables

Normally, member variables are not initialized using assignments in the constructor body. Instead, they are initialized in the constructor *initializer list*:

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

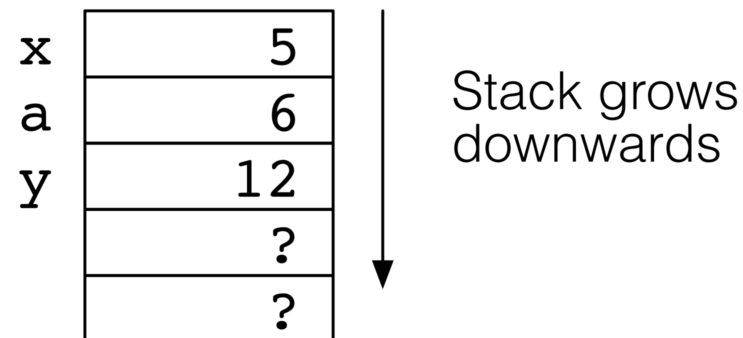
- Syntax: `member-variable(initializer-expression)`
- It is usually better to use the initializer list, and sometimes necessary. Discussed further later.

Stack, Heap, Activation Records

The (virtual) memory for a program is divided into two areas: the *stack* for local variables in functions, the *heap* for dynamically allocated variables.

When a function is entered, memory is allocated on the stack for parameters and local variables. This memory area is called an *activation record* or *stack frame*. When the function is exited, the activation record is deallocated.

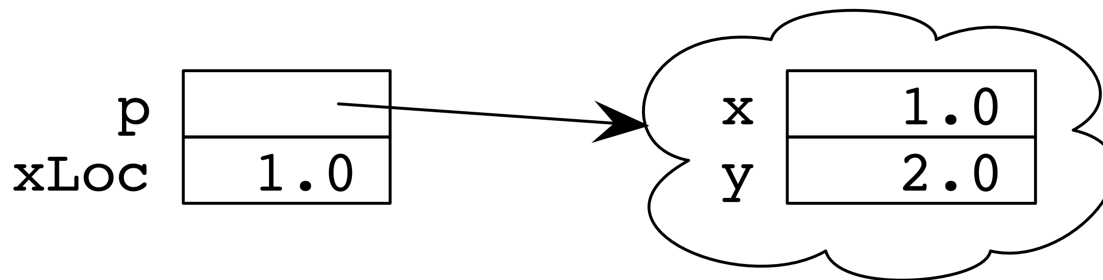
```
int main() {
    int x = 5;
    f(x + 1);
}

void f(int a) {
    int y = 2 * a;
}
```



Objects in Java

A Java object is always allocated on the heap, and you need a pointer (reference variable) to keep track of the object:

```
void f() {
    Point p = new Point(1, 2);
    double xLoc = p.getX();
    ...
}
```



When a Java object can no longer be reached (no references point to the object) it becomes garbage. The garbage collector reclaims memory used by unreachable objects.

Objects in C++, Stack Allocation

In C++, objects can be allocated in several different ways. The two most common are:

- stack allocated (automatic) — like any local variable, accessed by name
- heap allocated — like in Java, accessed through pointer

Stack allocation example:

```
void f() {
    int a = 5;
    Point p1(1, 2);
    int b = 6;
    double xLoc = p1.getX();
    ...
}
```

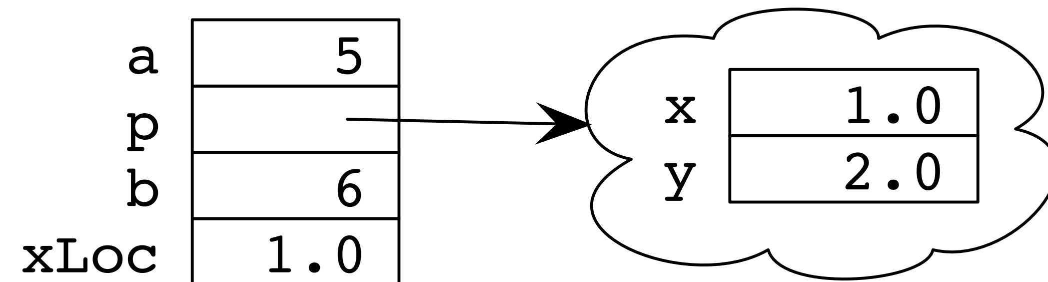
a	5
p1.x	1.0
p1.y	2.0
b	6
xLoc	1.0

`p1` (and `a`, `b`, and `xLoc`) are deallocated automatically when `f` is exited.

Heap Allocation

An object on the heap is similar to a Java object, but in C++ there is no garbage collector so you must explicitly delete the object.

```
void f() {
    int a = 5;
    Point* p = new Point(1, 2);
    int b = 6;
    double xLoc = p->getX();
    ...
    delete p;
}
```



`Point* p` means that `p` is a pointer, `p->` is used to access members in the object. The lifetime of a heap-allocated object is between `new` and `delete`; it has nothing to do with the block structure.

If you forget to delete an object, the result is a *memory leak*.

Safe Pointers C++11

In C++11, there are new library classes that encapsulate “raw” pointers and make sure that dynamic objects always are deallocated. The two most important:

`unique_ptr` A pointer which “owns” its object. If a pointer is reassigned, ownership is transferred.

`shared_ptr` A pointer which counts the number of references to the object. The object is deleted when it has no more references.

Do Not Use Dynamic Memory

Java programmers that come to C++ often continue programming in the Java style. This means, for example, that they create all objects with `new Classname`. This is seldom necessary.

It is only necessary to allocate an object on the heap (with `new`) if the object shall be shared, if the size of the object is unknown, or if the object must outlive the current scope. Examples:

- Implement a dynamic vector (like a Java `ArrayList`), or a dynamic string (like a Java `String` or `StringBuilder`), or ... There are library classes for such things.
- Implement a linked list (where the nodes are created in a member function), or ... There are library classes for such things.
- Handle polymorphic objects, handle objects that must be shared, ...

A Class Containing Objects

Implement a `Line` class that uses `Point` objects (slide 10) to represent the start and end points:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    ...
};
```

The following slides show alternative implementations. The variants with pointers are for illustration only, there is no reason to use dynamic memory in this class.

Class Line, Embedded Objects

Define the Point objects as member variables in the line class. Note *not* pointers to the objects:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    Point start;
    Point end;
};

Line::Line(double x1, double y1, double x2, double y2) :
    start(x1, y1), end(x2, y2) {}
```

- `start(x1, y1)` and `end(x2, y2)` are constructor calls which initialize the `Point` objects.
- No `delete`, no dynamic memory is involved.

Class Line, Dynamic Objects, C++98 version

Define "raw" pointers to the Point objects as member variables:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    Point* start;
    Point* end;
};

Line::Line(double x1, double y1, double x2, double y2) :
    start(new Point(x1, y1)), end(new Point(x2, y2)) {}
```

- Problem: the Point objects are never deleted, so there is a memory leak.

Destructors

The `Line` class needs a *destructor* which deletes the `Point` objects:

```
class Line {
public:
    ...
    ~Line();
private:
    Point* start;
    Point* end;
};

Line::~Line() {
    delete start;
    delete end;
}
```

- The destructor is called automatically when the object is destroyed and is responsible for cleaning up after the object.

Class Line, Dynamic Objects, Safe Pointers C++11

Define safe *pointers* to the `Point` objects as member variables:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    std::unique_ptr<Point> start;
    std::unique_ptr<Point> end;
};

Line::Line(double x1, double y1, double x2, double y2) :
    start(new Point(x1, y1)), end(new Point(x2, y2)) {}
```

- The dynamic objects are created in the constructor and handed over to the safe pointers.
- No visible `delete`, the safe pointers delete the objects when the pointers are destroyed (when the `Line` object is destroyed).

Inheritance

Inheritance in C++ is similar to Java inheritance, but member functions are by default not polymorphic. You must specify this with the keyword `virtual`.

```
class A {
public:
    void f();
    virtual void g();
};

class B : public A {
public:
    void f();
    virtual void g() override;
};
```

- `override` C++11 is not mandatory but recommended.

Calling Member Functions

```
void f() {
    A* pa = new A;
    pa->f(); // calls A::f
    pa->g(); // calls A::g

    A* pb = new B;
    pb->f(); // calls A::f, f is not virtual
    pb->g(); // calls B::g, g is virtual

    ...
    delete pa; // should have used unique_ptr ...
    delete pb;
}
```

- Non-virtual: the type of the *pointer* determines which function is called.
- Virtual: the type of the *object* determines which function is called.

Namespaces

Namespaces are used to control visibility of names (same purpose as Java packages). You can define your own namespaces. In a file *graphics.h:*

```
namespace Graphics {
    class Point { ... };
    class Line { ... };
}
```

In another file:

```
#include "graphics.h"

int main() {
    Graphics::Point p(1, 2);
    ...
}
```

Using Names from Namespaces

There are several ways to access names from namespaces (similar to importing from Java packages):

- Explicit naming:

```
Graphics::Point p(1, 2);
```

- `using` declarations:

```
using Graphics::Point;    // "import Graphics.Point"
Point p(1, 2);
```

- `using` directives:

```
using namespace Graphics; // "import Graphics.*"
Point p(1, 2);
```

Generic Programming

Java has generic classes:

```
ArrayList<Integer> v = new ArrayList<Integer>();
v.add(5);
int x = v.get(0);
```

C++ has generic (template) classes:

```
vector<int> v;
v.push_back(5);
int x = v[0];
```

- In Java, template parameters are always classes; may be of any type in C++. An `ArrayList` always holds objects, even if we use autoboxing and autounboxing as above.

Operator Overloading

In Java, operator + is overloaded for strings (means concatenation). In C++, you can define your own overloads of existing operators.

```
class Complex {
public:
    Complex(double re, double im);
    Complex operator+(const Complex& c) const;
private:
    double re;
    double im;
};

Complex Complex::operator+(const Complex& c) const {
    return Complex(re + c.re, im + c.im);
}
```

Now, you can add two complex numbers with `c1 + c2`.

General Program Structure

A main program uses a class `Stack`. The program consists of three compilation units in separate files:

```
class Stack {...};      // class definition, in file
                        // stack.h (header file)
Stack::Stack() { ... }  // member function definitions,
                        // in file stack.cc
int main() { ... }      // main function, in file example.cc
```

It should be possible to compile the *.cc* files separately. They need access to the class definition in *stack.h*, and use an `include` directive for this:

```
#include "stack.h"      // own unit, note " "
#include <iostream>     // standard unit, note < >
```

stack.h

```
#ifndef STACK_H // include guard, necessary to avoid
#define STACK_H // inclusion more than once
#include <cstddef> // for size_t
class Stack {
public:
    Stack(std::size_t sz);
    ~Stack();
    void push(int x);
    ...
private:
    int* v; // pointer to dynamic array
    std::size_t size; // size of the array
    std::size_t top;  // stack pointer
};
#endif
```

- This file is only included, not compiled separately.
- Don't write `using namespace` in a header file. A file that includes the header file may not be prepared for this.

stack.cc

```
#include "stack.h" // for class Stack

using namespace std; // ok here, this file will not be included

Stack::Stack(size_t sz) : v(new int[sz]), size(sz), top(0) {}

Stack::~Stack() { delete[] v; }

void Stack::push(int x) { v[top++] = x; }

...
```

This file can be compiled separately:

```
g++ -c stack.cc      // generates object code in stack.o
```

example.cc

```
#include "stack.h"   // for class Stack
#include <iostream>  // for cin, cout, ...

using namespace std; // ok here, this file will not be included

int main() {
    Stack s(100);
    s.push(43);
    ...
}
```

Compilation, linking and execution:

```
g++ -c example.cc                   // generates example.o
g++ -o example example.o stack.o // generates example
./example
```

Data Types, Expressions, and Such

- Data types
- Strings, vectors, arrays
- Expressions
- Statements
- Functions

Primitive Data Types

Character `char`, usually one byte; `wchar_t`, `char16_t`, `char32_t` `C++11`, wide characters.

Integral `int`, often 32 bits; `short`, 16 or 32 bits; `long`, 32 or 64 bits; `long long` `C++11` at least 64 bits.

Real `double`, often 64 bits; `float`, 32 bits; `long double` `C++11` often 80 bits.

Boolean `bool`, `true` or `false`; also see next slide.

- Sizes are implementation defined.
- The operator `sizeof` gives the size of an object or a type in character units: `sizeof(char) == 1` always, `sizeof(int) == 4` typically.
- Integral types and characters may be `signed` or `unsigned`. Implementation defined if a plain `char` is signed or unsigned.

Booleans

In early C++ standards, `int`'s were used to express boolean values: the value zero was interpreted as `false`, a non-zero value as `true`. There are still implicit conversions between booleans and integers/pointers. The following is correct C++, but shouldn't be used in new programs:

```
int a;
cin >> a;
if (a) { ... }      // equivalent to if (a != 0)

Point* p = list.getFirstPoint();
while (p) { ... }  // equivalent to while (p != 0)
                   // 0 is C++98 for 'nullptr'
```

Variables and Constants

Rules for declaration of variables are as in Java:

```
int x;            // integer variable, undefined. The compiler
                  // isn't required to check the use of
                  // uninitialized variables
double y = 1.5; // double variable, initial value 1.5
```

Constant values:

```
const int NBR_CARDS = 52;
```

Variables may be defined outside of functions and classes; then they become globally accessible. Use with great care.

Enumerations

An enumeration is a type which defines a set of constant names. The new standard introduced "scoped enums" `C++11`, where the constant names are local to an enum (similar to constants in a class). Example:

```
enum class open_mode {INPUT, OUTPUT, APPEND};
```

An object of an enumeration type may only be assigned one of the enumerator values:

```
int open_file(const string& filename, open_mode mode) {
    ...
}

int file_handle = open_file("input.txt", open_mode::INPUT);
```

References

A reference is an alternative name for an object, an "alias". It is *not* the same as a Java reference (which is a pointer). References are mainly used as function parameters and as function return values.

A reference must be initialized when it's created, and once initialized it cannot be changed. There are no "null references".

```
int ival = 1024;     // a normal int
int& refval = ival; // refval refers to ival
refval += 2;         // adds 2 to ival
int ii = refval;     // ii becomes 1026
```

A `const` reference may refer to a `const` object, a temporary object or an object needing conversion. Not much use for this now, becomes important later.

```
const int& r1 = 42;
const double& r2 = 2 * 1.3;
const double& r3 = ival;
```

Pointers

A pointer points to a memory location and holds the address of that location. Unlike Java, a pointer may point to (almost) anything: a variable of a basic type like `int`, a stack-allocated variable, ... A pointer is declared with a `*` between the type and the variable name:

```
int* ip1;      // can also be written int *ip1
Point* p = nullptr; // pointer to object of class Point.
               // nullptr is "no object", same as Java's null.
               // nullptr is new in C++11, C++98 used 0
```

One way (not the most common; the most common is to use `new` to create a dynamic variable on the heap) of obtaining a pointer value is to precede a variable with the symbol `&`, the *address-of* operator:

```
int ival = 1024;
int* ip2 = &ival;
```

Dereferencing Pointers

The contents of the memory that a pointer points to is accessed with the * operator. This is called *dereferencing*.

```
int ival = 1024;
int* ip2 = &ival;
cout << *ip2 << endl; // prints 1024
*ip2 = 1025;
```

Anything can happen if you dereference a pointer which is undefined or null. Comparing pointers with references:

- Pointers must be dereferenced, not references
- Pointers can be undefined or null, not references
- Pointers can be changed to point to other objects, not references

Type Aliases

A name may be defined as a synonym for an existing type name. Traditionally, `typedef` is used for this purpose. In the new standard, an alias declaration can also be used `C++11`. The two forms are equivalent.

```
using newType = existingType; // C++11
typedef existingType newType; // equivalent, still works
```

Examples:

```
using counter_type = unsigned long;
using table_type = std::vector<int>;
```

The auto Type Specifier C++11

Sometimes type names are long and tedious to write, sometimes it can be difficult for the programmer to remember the exact type of an expression. The compiler, however, has no problems to deduce a type. The `auto` keyword tells the compiler to deduce the type from the initializer. Examples:

```
vector<int> v; // a vector is like a Java ArrayList
...
auto it = v.begin(); // begin() returns vector<int>::iterator

auto func = [](int x) { return x * x; }; // a lambda function
```

Do *not* use auto when the type is obvious, for example with literals.

```
auto sum = 0; // silly, sum is int
```

Auto Removes & and `const`

When `auto` is used to declare variables, “top-level” & and `const` are removed. Examples:

```
int i = 0;
int& r = i;
auto a = r;  // a is int, the value is 0
auto& b = r; // b is ref to int

const int ci = 1;
auto c = ci;        // c is int, the value is 1
const auto d = ci; // d is const int, the value is 1
```

The `decltype` Type Specifier C++11

Sometimes we want to define a variable with a type that the compiler deduces from an expression, but do not want to use that expression to initialize the variable. `decltype` returns the type of its operand:

```
sometype f() { ... }

decltype(f()) sum = 0;

vector<Point> v;
...
for (decltype(v.size()) i = 0; i != v.size(); ++i) { ... }
```

- `f()` and `v.size()` are not evaluated.

Strings

As in Java, the string type is a standard class. Unlike Java, strings are modifiable (similar to Java's string builders/string buffers).

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

int main() {
   std::cout << "Type your name ";
   std::string name;
   std::cin >> name;
   std::cout << "Hello, " + name << std::endl;
}
```

- Overloaded operators: `<<`, `>>`, `+`, `<` , `<=`, ...
- Character access: `s[index]`, no runtime check on the index.
- Operations: `s.size()`, `s.substr(start, nbr)`, many more.

Looping Through a `string`

```
string s = ...;

for (string::size_type i = 0; i != s.size(); ++i) {
    cout << s[i] << endl;
    s[i] = ' ';
}
```

The new standard introduced a *range-based* `for` statement `C++11`:

```
for (auto& c : s) {
    cout << c << endl;
    c = ' ';
}
```

Note the reference: it is necessary since the characters are modified.

A Note About != and ++

For statements, Java and C++ style:

```
for (int i = 0; i < 10; i++) { ... }  // Java

for (int i = 0; i != 10; ++i) { ... } // C++
```

In C++, testing for end of range with `!=` is preferred instead of `<`. And pre-increment (`++i`) is preferred instead of post-increment (`i++`). When the loop variable is of a built-in type, like here, both alternatives are equivalent. But the loop variable can be an object of a more complex type, like an iterator, and not all iterators support `<`. And pre-increment is more efficient than post-increment.

Vectors

The class `vector` is a template class, similar to Java's `ArrayList`. A vector can hold elements of arbitrary type. An example (read words and print them backwards):

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

using namespace std;
int main() {
    vector<string> v;
    string word;
    while (cin >> word) {
        v.push_back(word);
    }
    for (int i = v.size() - 1; i >= 0; --i) {
        cout << v[i] << endl;
    }
}
```

Vector Notes

- A vector is initially empty, elements are added at the end with `push_back`.
- Storage for new elements is allocated automatically.
- Element access with `v[index]`, no runtime check on the index.
- Elements cannot be added with subscripting: `vector<int> v; v[0] = 1` is wrong.
- The subscript type is `vector<T>::size_type`; usually this is `unsigned long`.
- Vectors may be copied: `v1 = v2`; and compared: `v1 == v2`.

Introducing Iterators

- `vector` is one of the container classes in the standard library. Usually, *iterators* are used the access the elements of a container (not all containers support subscripting, but all have iterators).
- An iterator "points" to one of the elements in a collection (or to a location immediately after the last element). It may be *dereferenced* with `*` to access the element to which it points and incremented with `++` to point to the next element.
- Containers have `begin()` and `end()` operations which return an iterator to the first element and to one past the end of the container.

Traversing a vector

```
vector<int> v;
...
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    *it = 0;
}
```

Better with `auto` C++11:

```
for (auto it = v.begin(); it != v.end(); ++it) {
    *it = 0;
}
```

Even better with a range-based for C++11:

```
for (int& e : v) {
    e = 0;
}
```

Iterator Arithmetic

Vector iterators support some arithmetic operations. Find the first negative number in the second half of the vector `v`:

```
auto it = v.begin() + v.size() / 2; // midpoint
while (it != v.end() && *it >= 0) {
    ++it;
}
if (it != v.end()) {
    cout << "Found at index " << it - v.begin() << endl;
} else {
    cout << "Not found" << endl;
}
```

const Iterators

A "normal" iterator can be used to read and write elements. A `const_iterator` may only be used to read elements. `cbegin()` and `cend()`, new in `C++11`, return const iterators.

```
vector<int> v;
...
for (auto it = v.cbegin(); it != v.cend(); ++it) {
    cout << *it << endl;
}
```

When the container is a constant object, `begin()` and `end()` return constant iterators. Example (more about parameters later):

```
void f(const vector<int>& v) {
    for (auto it = v.begin(); it != v.end(); ++it) {
        *it = 0; // Wrong -- 'it' is a constant iterator
    }
}
```

Arrays

- Bjarne Stroustrup: "The C array concept is broken and beyond repair."
- Modern C++ programs normally use vectors instead of built-in arrays.
- There are no checks on array subscripts during runtime (nothing like Java's `ArrayIndexOutOfBoundsException`). With a wrong index you access or destroy an element outside the array.
- There are two ways to allocate arrays: on the stack (next slides) and on the heap (after pointers).

Stack-Allocated Arrays

```
void f() {
    int a = 5;
    int x[3];  // size must be a compile-time constant
    for (size_t i = 0; i != 3; ++i) {
        x[i] = (i + 1) * (i + 1);
    }
    ...
}
```

Activation record:

a	5
x[0]	1
x[1]	4
x[2]	9
i	3

More on Arrays

- Array elements can be explicitly initialized:

```
int x[] = {1, 4, 9};
```

- String literals are character arrays with a null terminator:

```
char ca[] = "abc"; // ca[0] = 'a', ..., ca[3] = '\0'
```

- Arrays cannot be copied with the assignment operator or compared with the comparison operators.
- In most cases when an array name is used, the compiler substitutes a pointer to the first element of the array:

```
int* px1 = x;
int* px2 = &x[0]; // equivalent
```

Pointers and Arrays

You may use a pointer to access elements of an array, and use pointer arithmetic to add to/subtract from a pointer. Pointers are iterators for arrays.

```
int x[] = {0, 2, 4, 6, 8};
for (int* px = x; px != x + 5; ++px) {
    cout << *px << endl;
}
```

When a pointer is incremented, the increments are in the size of the addressed data type, so `px + 1` means `px + sizeof(T)` for an array of type `T`. You may also subtract two pointers to get the number of array elements between the pointers.

Accessing array elements with subscripting or through pointers is equivalent. Actually, subscripting is defined as follows:

```
x[index] <=> *(x + index)
```

Library begin and end Functions C++11

In the example on the previous slide we obtained a pointer to the first element with the array name, a pointer past the last element with the array name plus the number of elements. For a `vector`, we would have used the `begin` and `end` members.

Since an array is a built-in type it doesn't have any member functions. Instead, there are library functions for this purpose:

```
int x[] = {0, 2, 4, 6, 8};
for (int* px = begin(x); px != end(x); ++px) {
    cout << *px << endl;
}
```

The global `begin` and `end` functions can be used also for other containers.

C-Style Strings

A C-style string is a null-terminated array of characters. In C++, C-style strings are almost only used as literals: `"asdfg"`. The C library `<cstring>` contains functions to manipulate C-style strings, for example `strlen` (string length), `strcpy` (string copy), `strcmp` (string compare). These functions are difficult to use correctly, since the caller is responsible for allocating storage for the strings.

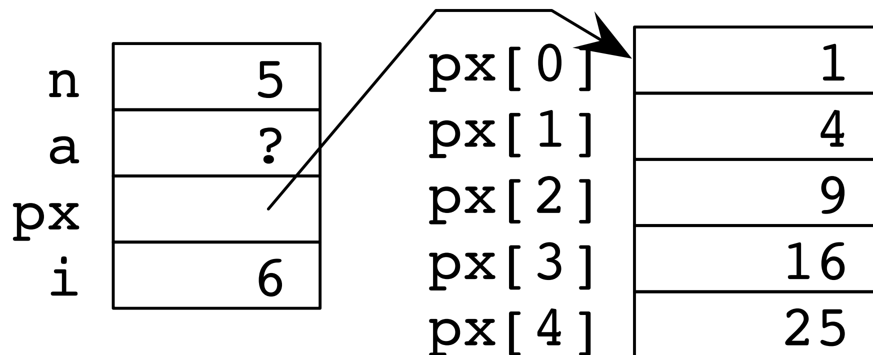
C++ strings and C-style strings can co-exist. In class `string`, most functions and operators are overloaded to accept C-style strings as parameters. Examples:

```
const char* cs = "asdfg";
string s = cs;
s += ".";
string::size_type pos = s.find("df");
```

Heap-Allocated Arrays

Arrays on the heap are similar to Java arrays, but you must delete the array after use. Example:

```
void g(size_t n) {
    int a;
    int* px = new int[n]; // size may be dynamic, >= 0
    for (size_t i = 0; i != n; ++i) {
        px[i] = (i + 1) * (i + 1);
    }
    ...
    delete[] px; // note []
}
```



Array Notes

- A heap-allocated array must be accessed via a pointer.
- A heap-allocated array does not contain information about its length.
- The global `begin()` and `end()` iterator functions cannot be used for heap-allocated arrays.
- `delete[]` is necessary to delete the array (otherwise the objects in the array will not be destroyed).

Operators

Most C++ operators are identical to the corresponding Java operators:

Arithmetic: `* / % + -`

Relational: `< <= = >= > !=`

Logical: `! && ||`

Bitwise: `&` bitwise and; `^` bitwise exclusive or; `|` bitwise inclusive or, `~` complement
`<<` left shift; `>>` right shift (implementation defined how the sign bit is handled, there is no `>>>` in C++)

Special: `? :`
`+= -= ...`
`++x x++ --x x--`

The Arrow Operator

To access a member in an object the dot operator is used. If you have a pointer to the object you use the arrow operator.

```
Point p1(1, 2);
Point* p = new Point(1, 2);
double xLoc1 = p1.getX();
double xLoc2 = p->getX();
```

Instead of using the arrow operator you could first dereference the pointer to get the object pointed to, then use the dot operator: `double xLoc2 = (*p).getX()`. This is difficult to read and shouldn't be used.

The `sizeof` Operator

`sizeof(type name)` returns the size of an object of the type, in character units. `sizeof(expression)` returns the size of the result type of the expression. Compare:

```
int a[5];
cout << sizeof(int) << endl; // 4 (if an int has 4 bytes)
cout << sizeof(a) << endl;   // 20, 4*5

int* pa = new int[5];
cout << sizeof(pa) << endl;  // 8 (the size of the pointer)
cout << sizeof(*pa) << endl; // 4 (points to int)

vector<int> v = { 1, 2, 3, 4, 5 };
cout << sizeof(v) << endl;   // 24 (implementation defined,
                             // size of the object)
```

`sizeof` cannot be used to retrieve the number of elements of a heap-allocated array or of a vector. (But can use `vector::size()`.)

Type Conversions

In Java, there are implicit type conversions from “small” types to “large” types. To convert in the other direction, where precision is lost, an explicit cast is necessary.

```
double d = 35;    // d == 35.0; implicit
d = 35.67;
int x = (int) d; // x == 35; explicit
```

In C++, there are implicit conversions in both directions.

```
d = 35.67;
int x = d;       // x == 35; implicit
```

You should use an explicit cast instead:

```
int x = static_cast<int>(d);
```

Different Casts

The C++ rules for implicit type conversions are complex, so it is best to always use explicit casts. There are other casts besides `static_cast`:

`dynamic_cast<type>(pointer)` for "downcasting" in an inheritance hierarchy.

`const_cast<type>(variable)` removes "constness" from a variable. Not often used.

`reinterpret_cast<type>(expr)` converts anything to anything else, just reinterprets a bit pattern. Only used in low-level systems programming.

The C (and Java) style of casting, `(int) d`, is also allowed in C++ but should *not* be used; depending on its argument it functions as a `const_cast`, `static_cast`, or a `reinterpret_cast`.

Numeric Conversions

A string of digits cannot be converted to a binary number with a cast, nor can a binary number be converted to a string. In the old standard, "string streams" were used for this purpose:

```
string s1 = "123.45";
istringstream iss(s1); // similar to a Java Scanner
double nbr;
iss >> nbr;
```

The new standard introduced conversion functions `C++11`:

```
string s1 = "123.45";
double nbr = stod(s1); // "string to double"
...
string s2 = to_string(nbr + 1);
```

Statements

No surprises here:

- `if`, `switch`, `while`, `do while`, `for`.
- The new standard has range-based `for` `C++11`.
- `break` and `continue`.
- Labeled `break` and `continue` are not available in C++ (you can use `goto` instead, but don't).

Exceptions

Throwing and catching exceptions:

- Anything can be thrown as an exception: an object, a pointer, an `int`, ..., but you usually throw an object (*not* a pointer to an object).
- There are standard exception classes: `exception`, `runtime_error`, `range_error`, `logic_error`, ...
- In an exception handler, the exception object should be passed by reference (`catch (exception& e)`).
- If an exception isn't caught, the library function `terminate` is called. It aborts the program.
- No Java-style "throws"-specification.

Exceptions, example

```
throw runtime_error("Wrong parameter values!");
throw 20;
```

```
try {
   // program statements
} catch (runtime_error& err) {
   // error handling code
} catch (...) {
   // default error handling code
}
```

Functions

Functions in C++ are similar to Java methods, but C++ functions may also be defined outside classes ("free", "stand-alone", or "global", functions).

Functions may be defined as `inline`. It means that calls of the function are replaced with the body of the function.

```
inline int sum(int i, int j) {
    return i + j;
}
```

`inline` is just a request to the compiler to generate inline code — the compiler may choose to ignore it. Inline function definitions must be placed in header files.

`constexpr` Functions

In some cases, for example when specifying array bounds, a constant expression is necessary. A function that is specified `constexpr` is evaluated during compilation if possible, and then it generates a constant expression. Example:

```
constexpr int sum(int i, int j) {
    return i + j;
}

void f(int n) {
    int x[sum(4, 5)]; // correct, equivalent to x[9]
    int y[sum(n, 5)]; // wrong, array bound isn't constant
}
```

`constexpr` functions are implicitly inline.

Local Static Variables

A class attribute in C++ (as in Java) may be `static`. This means that there is only one variable, which is shared between all objects of the class.

In C++, also functions may have static variables. Such a variable is created on the first call of the function, and its value is remembered between successive calls.

Count the number of calls of a function:

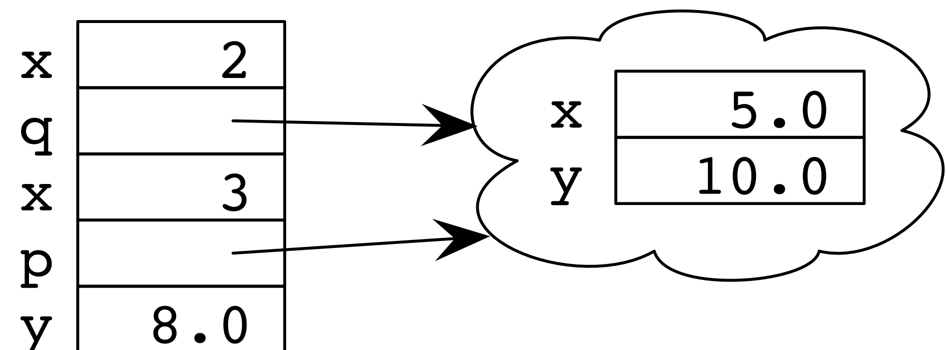
```
void f() {
    static int counter = 0;
    counter++;
    ...
}
```

In C, local static variables are essential. In C++, most functions are class members and member variables are used to remember values between function calls.

Argument Passing

Arguments to functions are by default passed by value, as in Java. This means that the value of the argument is computed and copied to the function, and that an assignment to the formal parameter does not affect the argument.

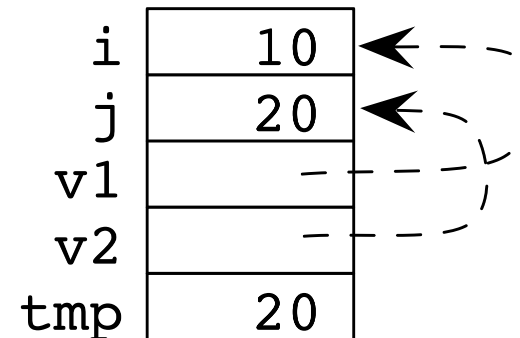
```
void f(int x, const Point* p) {
    x++;
    double y = x + p->getX();
}

int main() {
   int x = 2;
   Point* q = new Point(5, 10);
   f(x, q);
   ...
   delete q;
}
```



Reference Parameters

If you wish to change the value of an argument you must pass the argument by reference. Example:

```
void swap(int& v1, int& v2) {
    int tmp = v2;
    // see figure for memory state here
    v2 = v1;
    v1 = tmp;
}

int main() {
    int i = 10;
    int j = 20;
    swap(i, j);
}
```



How They Do It in C

C doesn't have references, so to write the `swap` function in C you must use pointers. This should be avoided in C++.

```
void swap(int* v1, int* v2) {
    int tmp = *v2;
    *v2 = *v1;
    *v1 = tmp;
}

int main() {
    int i = 10;
    int j = 20;
    swap(&i, &j);
}
```

Constant Pass by Reference

Pass by reference is also used when you have a large object that you don't wish to copy on each call.

```
bool isShorter(const string& s1, const string& s2) {
    return s1.size() < s2.size();
}
```

Note that `const` is essential (if you don't wish to modify the object):

- Without `const`, you could not pass a constant string as argument, or an object needing conversion (for example `isShorter(s, "abc")`).
- Without `const`, you could inadvertently change the state of the object in the function.

Parameters — Pointer or Reference

If a program uses stack-allocated objects, which should be the normal case, write functions with reference parameters. If the program must use heap-allocated objects, write functions with pointer parameters.

```
void refPrint(const Point& p) {
    cout << p.getX() << " " << p.getY() << endl;
}

void ptrPrint(const Point* p) {
    cout << p->getX() << " " << p->getY() << endl;
}

void f() {
    Point p1(10, 20);
    Point* pptr = new Point(30, 40);
    refPrint(p1);     // normal usage
    ptrPrint(pptr);   // also normal usage
    ...
    delete pptr;
}
```

Pointer or Reference, cont'd

You *can* pass an object even if you have a pointer, or a pointer if you have an object, but that should be avoided:

```
void refPrint(const Point& p) { ... }

void ptrPrint(const Point* p) { ... }

void f() {
    Point p1(10, 20);
    Point* pptr = new Point(30, 40);
    refPrint(*pptr); // possible, but avoid
    ptrPrint(&p1);   // also possible, also avoid
    ...
    delete pptr;
}
```

Array Parameters

An array argument is passed as a pointer to the first element of the array. The size of the array must also be passed as an argument.

```
void print(const int* ia, size_t n) {
    for (size_t i = 0; i != n; ++i) {
        cout << ia[i] << endl;
    }
}

int main() {
    int[] a = {1, 3, 5, 7, 9};
    size_t size = sizeof(a) / sizeof(*a);
    print(a, size);
}
```

Note that `const int*` is a variable pointer to a constant integer. A constant pointer to a variable integer is written `int* const`.

Another Way To Pass Array Parameters

Another way of passing array arguments is inspired by iterators. You pass pointers to the first and one past the last element:

```
void print(const int* beg, const int* end) {
    for (; beg != end; ++beg) {
        cout << *beg << endl;
    }
}

int main() {
    int[] a = {1, 3, 5, 7, 9};
    size_t size = sizeof(a) / sizeof(*a);
    print(a, a + size); // or print(begin(a), end(a))
}
```

Functions with Varying Parameters C++11

A function with an unknown number of arguments of the same type can pack the arguments in an `initializer_list` object. An `initializer_list` has `begin()` and `end()` operations. Example:

```
class MyVector {
public:
    MyVector(const initializer_list<int>& il)
            : v(new int[il.size()]), size(il.size()) {
        size_t i = 0;
        for (auto x : il) { v[i++] = x; }
    }
private:
    int* v;
    int size;
};

MyVector v1({1, 2, 3, 4});
MyVector v2 = {5, 6, 7, 8};
```

Functions Returning Values

The value that a function returns is copied from the function to a temporary object at the call site.

```
string strip(const string& s) {
    string ret;
    for (char c : s) {
        if (c != ' ') {
            ret += c;
        }
    }
    return ret; // the return value is copied
}

int main() {
    string s = strip("Mary had a little lamb") + ".";
    ...
}
```

Functions Returning References

A function can return a reference, and if it does, it can be called on either side of a = operator. This is mainly used when writing operator functions that overload standard operators, e.g., assignment or subscripting. The following example would have made more sense if the array had been a class attribute and `element` had been an overloaded `operator[]`.

```
int& element(const int* x, size_t i) {
    // ... could insert index check here
    return x[i];
}

void f() {
    int squares[] = {1, 4, 9, 15, 25};
    element(squares, 3) = 16;
    ...
}
```

Reference and Pointer Pitfalls

Never return a reference or a pointer to a local object. The return value will refer to memory that has been deallocated and will be reused on the next function call.

```
const string& strip(const string& s) { // note & on the return type
    string ret;
    ...
    return ret; // a reference to the local variable is copied
}

int main() {
    string s = strip("Mary had a little lamb") + ".";
    ... disaster
}
```

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");
```

new and delete

Usage of the operators `new` and `delete` are translated into calls of the following functions, which allocate/deallocate raw memory:

```
void* operator new(size_t bytes);
void operator delete(void* p) noexcept;
```

Example:

```
Point* p = new Point(10, 20);
// allocate raw memory, initialize the object
// Point* p = static_cast<Point*>(::operator new(sizeof(Point)));
// p->Point::Point(10, 20);

delete p;
// destruct the object, free memory
// p->~Point();
// ::operator delete(p);
```

Allocators

As seen on the previous slide, `new` combines memory allocation with object construction, and `delete` combines destruction with deallocation. This is true also for arrays of objects: `new Point[10]` default-constructs 10 objects.

Now consider implementing the library class `vector`. When `push_back(p)` is executed on a full vector and a new chunk of memory must be allocated, no objects should be created. Instead, `p` should be copied (or moved) into raw memory.

The library class `allocator` allocates and deallocates raw memory:

```
allocator<Point> alloc;
auto p = alloc.allocate(10); // allocates space for 10
                             // unconstructed Points,
                             // returns a pointer
...
alloc.deallocate(p, 10);     // deallocate
```

Constructing and Destroying Objects

When raw memory has been allocated, objects can be constructed in that area. Before memory is deallocated, the objects must be destroyed.

```
allocator<Point> alloc;
auto p = alloc.allocate(10);

auto q = p; // where to start constructing objects
alloc.construct(q++, 10, 20); // Point(10, 20)
alloc.construct(q++, 30, 40); // Point(30, 40)
...
while (q != p) {
    alloc.destroy(--q); // execute ~Point() for each object
}

alloc.deallocate(p, 10);
```

Reference Semantics

In Java, all objects are manipulated via reference variables (pointers in C++). You may assign pointers to pointer variables but assignments never touch the "inside" of the objects.

```
Car* myCar = new Car("ABC123");
Car* yourCar = new Car("XYZ789");
...
delete myCar;        // my car to the scrapheap
myCar = yourCar;     // I buy your car
yourCar = nullptr; // you don't own a car now
```

This is called *reference semantics*, and it is often the preferred way to manipulate objecs.

Value Semantics

Reference semantics is not always what you want. Consider manipulating strings via pointers:

```
string* s1 = new string("asdfg");
string* s2 = s1;     // two pointers to the same object

s1->erase(0, 1);     // erase the first character of s1
cout << *s1 << endl; // prints sdfg
cout << *s2 << endl; // also prints sdfg, maybe unexpectedly
                     // since you haven't touched s2
```

In C++ you may copy objects, not only pointers, to get the desired results. This is called *value semantics*.

```
string s1 = "asdfg";
string s2 = s1;      // copy the object
s1.erase(0, 1);      // only s1 is modified
```

Which Semantics is Best?

Value semantics is preferred for “small”, “anonymous”, objects (numbers, strings, dates, colors, ...). When object *identity* is important, reference semantics is better. Example:

```
Car myCar("ABC123");
Car yourCar("XYZ789");

myCar = yourCar; // copy the object => two cars that are
                 // identical. Not reasonable!
```

Cars have identity, so they should be manipulated using reference semantics. However, in C++ value semantics is sometimes used also for such classes.

Copying Objects

Objects are copied in several cases:

- Initialization

```
Person p1("Bob");
Person p2(p1);
Person p3 = p1;
```

- Assignment

```
Person p4("Alice");
p4 = p1;
```

- Parameter by value

```
void f(Person p) { ... }
f(p1);
```

- Function value return

```
Person g() {
    Person p5("Joe");
    ...
    return p5;
}
```

Initialization and Assignment

The following statements may look similar, but they are handled by different C++ language constructs:

```
Person p1("Bob");
Person p3 = p1; // initialization

Person p4("Alice");
p4 = p1;         // assignment
```

Initialization
: A new, uninitialized object is initialized with a copy of another object. This is handled by the *copy constructor* of the class, `Classname(const Classname&)`. It also handles value parameters and function value return.

Assignment
: An existing object is overwritten with a copy of another object. This is handled by the *assignment operator* of the class, `Classname& operator=(const Classname&)`.

The Synthesized Copy Functions

In your own classes, you can implement the copy constructor and assignment operator to get the desired copy behavior. If you don't do this, the compiler synthesizes a copy constructor and an assignment operator which perform memberwise ("shallow") copying.

You should *not* write your own copy constructor or assignment operator in a class that doesn't use dynamic resources. Example:

```
class Person {
public:
    // this is the copy constructor that the compiler
    // creates for you, and you cannot write a better one
    Person(const Person& p)
        : name(p.name), age(p.age) {}
private:
    string name;
    unsigned int age;
};
```

Preventing Copying C++11

Some classes shouldn't allow objects of the class to be copied. This is achieved by declaring the copy constructor and the assignment operator as `delete`-d:

```
class Person {
public:
    ...
    Person(const Person&) = delete;
    Person& operator=(const Person&) = delete;
private:
    ...
};
```

In C++98, the same effect was achieved by making the copy constructor and assignment operator `private`.

A String Class

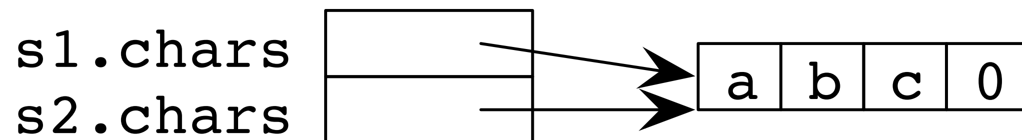
As an example of a class that allocates and deallocates dynamic memory we will use a simplified string class. The class should have value semantics. The characters are stored in a C-style string (a null-terminated `char` array). The `<cstring>` function `strlen` computes the number of characters in a C-style string, the `function` strcpy copies the characters.

```
class String {
public:
    String(const char* s) : chars(new char[strlen(s) + 1]) {
        strcpy(chars, s); // copy s to chars
    }
    ~String() { delete[] chars; }
private:
    char *chars;
};
```

Without a Copy Constructor, I

We haven't written a copy constructor for `String`, so the constructor generated by the compiler is used. It performs a memberwise copy.

```
void f() {
    String s1("abc");
    String s2 = s1; // see figure for memory state after copy
}
```



When the function exits:

1. The destructor for `s2` is called, `s2.chars` is deleted.
2. The destructor for `s1` is called, `s1.chars` is deleted.
3. This is disaster — you must not delete the same object twice.

Without a Copy Constructor, II

Value parameter example:

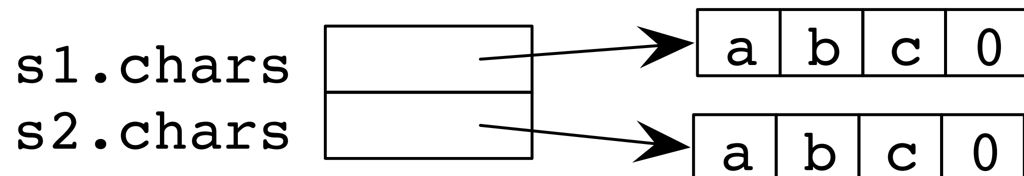
```
void f(String s) {
    ...
}

void g() {
    String s1("abc");
    f(s1);
}
```

1. In the call `f(s1)`, `s1` is copied to `s`. Now, `s.chars` and `s1.chars` point to the same array.
2. When `f` exits, the destructor for `s` is called, `s.chars` is deleted. Now, `s1.chars` points to deallocated memory and `s1` cannot be used.
3. When `g` exits, the destructor for `s1` is called and `s1.chars` is deleted ...

Defining a Copy Constructor

Class `String` must have a copy constructor that performs a *deep copy*, i.e., copies not pointers but what the pointers point to.

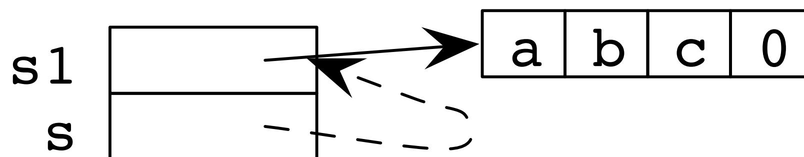
```
String(const String& rhs)
        : chars(new char[strlen(rhs.chars) + 1]) {
    strcpy(chars, rhs.chars);
}

void f() {
    String s1("abc");
    String s2 = s1; // see figure for memory state after copy
}
```



Avoiding Copy Problems

The copy constructor is never invoked if objects always are manipulated via pointers or references. For strings, this is not possible, since we need value semantics. But parameter copying can be avoided, and it doesn't happen if the parameter is passed by reference.

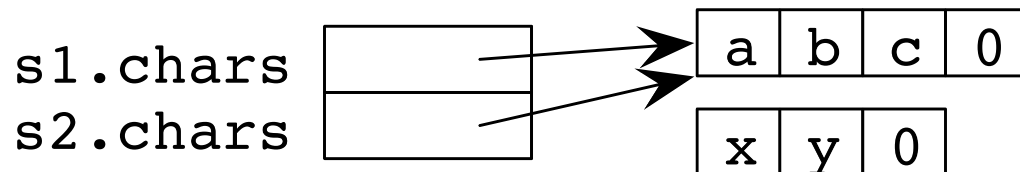
```
void f(const String& s) { ... }

void g() {
    String s1("abc");
    f(s1);
}
```



Assignment

Copy problems, with an extra complication, also occur when objects are assigned.

```
void f() {
    String s1("abc");
    String s2("xy");
    s2 = s1; // see figure for memory state after copy
}
```



The extra complication is that the array "xy" cannot be reached after the assignment, which leads to a memory leak. And as before, the destructor is called twice for the same object.

Overloading Assignment

To solve the assignment copying problems, you must define your own version of the assignment operator. This is done by defining an *operator function* `operator=`.

```
class String {
public:
    String& operator=(const String&);
    ...
};
```

With this operator function, the statement

```
s1 = s2;
```

is converted by the compiler into

```
s1.operator=(s2);
```

Implementing operator=

Implementation of `String::operator=`:

```
String& String::operator=(const String& rhs) {
    if (&rhs == this) {
        return *this;
    }
    delete[] chars;
    chars = new char[strlen(rhs.chars) + 1];
    strcpy(chars, rhs.chars);
    return *this;
}
```

Notice the similarities and dissimilarities to the copy constructor:

- Both functions perform a deep copy of the object's state.
- In addition, `operator=` must 1) delete the old state; 2) return the object being assigned to; 3) check for self-assignment. Explanations on next slide.

Details, operator=

1. Delete the old state (`delete[] chars`): this is necessary to prevent memory leaks.
2. Return the object (`return *this`): this is necessary to make it possible to chain assignments.

   ```
   s1 = s2 = s3; // compiled into s1.operator=(s2.operator=(s3))
   ```

   `operator=` returns a *reference* to the assigned object. This is safe, because the object certainly exists.
3. Check for self-assignment (`if (&rhs == this)`):

   ```
   s1 = s1;      // compiled into s1.operator=(s1)
   ```

   Here, `*this` and `rhs` are the same object. Without the check, you would first delete `s1.chars`, then allocate a new array, then copy from the uninitialized array.

Swapping Values

The library defines a function `swap` which swaps two values (as a template function that may be called with parameters of arbitrary types):

```
template <typename T>
inline void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

For the `String` class it is expensive to perform one copy and two assignments. See next slide for a better alternative.

Efficient Swapping

All you need to do when swapping two `String` objects is to swap the pointers to the character arrays:

```
class String {
    friend void swap(String& s1, String& s2);
public:
    ...
};

inline void swap(String& s1, String& s2) {
    using std::swap;
    swap(s1.chars, s2.chars);
}
```

Moving Objects

In the `String` class, the default copy constructor didn't work correctly:

```
String(const String& rhs) : chars(rhs.chars) {}
```

The copy constructor must perform a deep copy:

```
String(const String& rhs)
        : chars(new char[strlen(rhs.chars) + 1]) {
    strcpy(chars, rhs.chars);
}
```

If we are certain that the object that we're copying *from* will not be used after the copy, we could write the copy constructor like this:

```
String(String& rhs) : chars(rhs.chars) {
    rhs.chars = nullptr;
}
```

This is called a *move constructor*.

When Moving is Possible

But how can we (or rather the compiler) be "certain that the object that we're copying *from* will not be used after the copy?"

The answer is that *temporary values* can be moved from, since these will be destroyed after use. And the compiler can recognize temporary values:

```
String s1("abc");
String s2("def");
String s3 = s1 + s2; // the result of '+' is a temporary value

void f(String s);
f("abcd"); // => f(String("abcd")), the argument is a temporary

String g() {
    ...
    return ...; // the return value is a temporary
}
```

Lvalues and Rvalues, Rvalue References C++11

An lvalue is persistent (e.g. a variable). An rvalue is not persistent (e.g. a literal or a temporary). A regular non-const reference can bind only to an lvalue. In the new standard *rvalue references* have been introduced. An rvalue reference can bind only to an rvalue. Examples:

```
String s1("abc");
String s2("def");

String& sref = s1; // reference bound to a variable

String&& srr = s1 + s2; // rvalue reference bound to a temporary
```

You obtain an rvalue reference with `&&`.

Move Constructors

Now that we have rvalue references, the move constructor can be written. The copy constructor is still necessary for regular copying.

```
String(String&& rhs) noexcept : chars(rhs.chars) {
    rhs.chars = nullptr;
}

String(const String& rhs)
        : chars(new char[strlen(rhs.chars) + 1]) {
    strcpy(chars, rhs.chars);
}
```

`noexcept` C++11 informs the compiler that the constructor will not throw any exceptions. This is only important for move operations.

Move Assignment Operator

A class whose objects can be moved should also have a move assignment operator:

```
String& operator=(String&& rhs) noexcept {
    if (&rhs == this) {
        return *this;
    }
    delete[] chars;
    chars = rhs.chars;
    rhs.chars = nullptr;
    return *this;
}
```

Explicit Moving

Sometimes the programmer, but not the compiler, is certain that it is safe to move an object rather than copying it. The library function `std::move` returns an rvalue reference to its argument:

```
template <typename T>
inline void swap(T& a, T& b) {
    T temp = std::move(a); // a is moved to temp, a is empty

    a = std::move(b);      // but a isn't used, instead b is
                           // moved to a, b is empty

    b = std::move(temp);   // but b isn't used, instead temp is
                           // moved to b, temp is empty

}                          // but temp isn't used, instead
                           // destroyed
```

Canonical Construction Idiom

1. When a class manages a resource, it needs its own destructor, copy and move constructor, and copy and move assignment operator ("rule of five").
2. The constructor should initialize data members and allocate required resources.
3. The copy constructor should copy every element (deep copy). The move constructor should move instead, making the right-hand side empty.
4. The copy assignment operator should release resources that are owned by the object on the left-hand side of the assignment and then copy elements from the right side to the left side (deep copy). The move assignment operator should move instead, making the right-hand side empty.
5. The destructor should release all resources.

Vectors of Objects

As we have mentioned earlier, you should almost always use vectors instead of arrays. If you need to keep track of persons (objects of class `Person`), you must decide what to store in the vector: `Person` objects or pointers to `Person` objects.

```
vector<Person> v;  // vector of Person objects
Person p1("Bob");
v.push_back(p1);   // p1 is copied
v.push_back(Person("Alice")); // the object is moved
...
for (const auto& p : v) {
    cout << p.getName() << endl;
}
```

- Class `Person` must have a copy (and/or move) constructor.
- Large objects can be slow to copy.
- You cannot store objects of subclasses of `Person`.

Vectors of Pointers to Objects

You can instead store pointers to heap-allocated `Person` objects.

```
vector<Person*> pv;
Person* p = new Person("Bob");
pv.push_back(p);
...
for (auto pptr : pv) {
    cout << pptr->getName() << endl;
}
...
for (auto pptr : pv) {
    delete pptr;
}
```

- Only pointers are copied, no copy constructor is needed.
- You can store pointers to objects of subclasses of `Person`.
- But you must delete all objects before the vector goes out of scope.

Vectors of unique_ptrs C++11

You don't have to worry about deletion if you use unique pointers:

```
vector<unique_ptr<Person>> upv;
upv.push_back(unique_ptr<Person>(new Person("Bob")));
...
for (const auto& upptr : upv) {
    cout << upptr->getName() << endl;
}
```

- Same advantages as vector of raw pointers, but you don't have to worry about deletion.
- But you must keep in mind that unique pointers cannot be copied, just moved.
- Shared pointers can naturally also be used.

Arrays of Objects

Everything becomes much more difficult if you use arrays instead of vectors. Consider:

```
Person v[100]; // the size must be a compile-time constant,
               // the class must have a default constructor
```

If the size isn't known at compile time, the array must be created on the heap and later deleted:

```
size_t size = ...;
Person* pv = new Person[size];
...
delete[] pv;
```

The brackets on `delete` informs the compiler that an array is deleted, rather than a single object. The compiler generates code to call the destructor for each object in the array.

Dynamic Array of Pointers to Objects

If you use an array to store pointers to Person objects, and the array should be dynamic, you must do something like the following:

```
Person** pv = new Person*[size]; // pointer to first element of
                                 // array, which is a pointer
                                 // to a Person object
size_t nbrObj = 0;
pv[nbrObj++] = new Person("Bob");
...
for (size_t i = 0; i != nbrObj; ++i) {
    cout << pv[i]->getName() << endl;
}
...
for (size_t i = 0; i != nbrObj; ++i) {
    delete pv[i];
}
delete[] pv;
```

More Advanced Class Concepts

- Operator overloading
- Inheritance
- Templates

Operator Overloading

In most programming languages some operators are overloaded. For instance, the arithmetic operators are usually overloaded for integers and reals — different addition operations are performed in the additions `1 + 2` and `1.0 + 2.0`. The compiler uses the types of the operands to determine the correct operation.

In C++, overloading can be extended to user-defined types (classes). Thus, `a + b` can mean addition of complex numbers, addition of matrices, etc.

Rules for Overloading

- All operators can be overloaded except `::` (scope resolution), `.` (member selection), `.*` (member selection through pointer to member), `? :`, `sizeof`.
- It is not possible to invent new operator symbols or to change the priority between operators.
- At least one of the operands of an overloaded operator must be an object (*not* a pointer to an object). So you cannot redefine the meaning of operators for built-in types, e.g., `int + int`.

Operator Functions

Each usage of an overloaded operator is translated into a call of an *operator function*. We have already seen one example, `operator=`. Operator functions are "normal" functions with "strange names" like `operator=`, `operator+`, ...

Operator functions may be members or nonmembers. Rules:

- Assignment (`=`), subscript (`[]`), call (`()`), and member access arrow (`->`) *must* be members.
- Operators that change the state of their object should be members: `+=`, `++`, ...
- Operators that don't change the state of any object should be nonmembers: `+`, `*`, `==`, ...

Example, Class Integer

In the following, we will use a class `Integer` to demonstrate overloading. The class is merely a wrapper for integer values, but it could easily be extended, for example to perform modulo arithmetic. Or, add an attribute to create a `Complex` class.

```
class Integer {
public:
    Integer(int v = 0) : value(v) {}
private:
    int value;
};
```

The constructor is not explicit, so you can write `Integer a = 5` as well as `Integer a(5)`.

Input and Output

You should be able to read and write `Integer` objects just as other variables, like this:

```
Integer a;
cin >> a;  // compiled into operator>>(cin, a);
cout << a; // compiled into operator<<(cout, a);
```

- The functions `operator>>` and `operator<<` must be nonmembers (otherwise they would have to be members of the `istream`/`ostream` classes, and you cannot modify the library classes).
- The input stream should be an `istream` so input can come from a file stream or a string stream, the output stream should be an `ostream`.
- The functions must be friends of the `Integer` class, so they can access the private member `value`.

Implementing operator>>

In the file `integer.h`:

```
class Integer {
    friend istream& operator>>(istream& is, Integer& i);
    ...
};

istream& operator>>(istream& is, Integer& i);
```

In the file `integer.cc`:

```
istream& operator>>(istream& is, Integer& i) {
    is >> i.value;
    return is;
}
```

Comments on next slide.

Comments, operator>>

- The operator function returns a reference to the stream so operations can be chained:

```
cin >> a >> b; // operator>>(operator>>(cin, a), b)
```

- The stream parameter is passed by reference (it is changed by the function).
- The `Integer` object is passed by reference (it is changed by the function).

Implementing operator<<

```
class Integer {
    friend ostream& operator<<(ostream& os, const Integer& i);
    ...
};

ostream& operator<<(ostream& os, const Integer& i) {
    os << i.value;
    return os;
}
```

- The function returns a reference to the stream so operations can be chained (`cout << a << b`).
- The stream is passed by reference (it is changed by the function).
- The `Integer` object is passed by constant reference (it shouldn't be copied and not changed by the function).
- The function doesn't do any extra formatting.

Compound Assignment and Arithmetic Operators

The class `Integer` should implement the arithmetic operators (+, -, . . .), and to be consistent with other types also the compound assignment operators (+=, -=, . . .). The arithmetic operators don't change the state of any object and should be non-members, the compound assignment operators change the state and should be members.

Always start by implementing the compound assignment operators, then use them to implement the arithmetic operators.

Example:

```
Integer a, b, sum;
a += b;       // compiled into a.operator+=(b);
sum = a + b; // sum = operator+(a, b);
```

Implementing operator+=

```
class Integer {
public:
    Integer& operator+=(const Integer& rhs) {
        value += rhs.value;
        return *this;
    }
    ...
};
```

- Like `operator=`, the function returns a reference to the current object so operations can be chained (`a += b += c`).

Implementing operator+

```
Integer operator+(const Integer& lhs, const Integer& rhs) {
    Integer sum = lhs; // local non-constant object
    sum += rhs;        // add rhs
    return sum;        // return by value
}
```

- The function returns a new `Integer` object, *not* a reference (you cannot return a reference to a local object).
- The parameters are passed by constant reference (not copied and not changed). (Pass by value would be ok here since the objects are small, but we use constant reference for all objects to be consistent.)
- The function need not be a friend of `Integer` since it calls `operator+=`, which is a member function and has access to the private member `value`.

Increment, Prefix and Postfix

`++x` Increment `x`, return the modified value.

`x++` Make a copy of `x`, increment `x`, return the copy. Since you have to copy an object, `x++` is less efficient than `++x`.

```
class Integer {
public:
    Integer& operator++();   // prefix, ++x
    Integer operator++(int); // postfix, x++
    ...
};
```

- Both functions have the same name; the postfix form has a dummy parameter (just an indicator, not used in the function).
- The prefix function returns a reference to the incremented value.
- The postfix function makes a copy of the object and returns the copy.

Implementing `operator++`

```
Integer& Integer::operator++() {
    ++value;
    return *this;
}

Integer Integer::operator++(int) {
    Integer ret = *this;
    ++*this; // use the prefix operator to do the work
    return ret;
}
```

- There is no need to give the dummy parameter to the postfix function a name.
- We could just as well have written `++value` instead of `++*this` in the second function. But `++*this` has the same effect and is better if incrementing is more complicated.

Equality Operators

```
class Integer {
    friend bool operator==(const Integer& lhs, const Integer& rhs);
    ...
};

bool operator==(const Integer& lhs, const Integer& rhs) {
    return lhs.value == rhs.value;
}

bool operator!=(const Integer& lhs, const Integer& rhs) {
    return ! (lhs == rhs);
}
```

- Global functions since they don't change any state.
- If you implement one of these you should implement both. Implement one in terms of the other.

Subscripting

We use the class String from slide 123 for the example on subscripting.

```
class String {
public:
    ...
    char& operator[](size_t index);
    const char& operator[](size_t index) const;
private:
    char* chars;
};
```

- Note two versions, for non-const and const objects. Overloading on const is possible.

Implementing operator[]

```
char& String::operator[](size_t index) {
    return chars[index];
}

const char& String::operator[](size_t index) const {
    return chars[index];
}
```

- It is essential that the non-const version returns a reference, so subscripting can be used on the left-hand side of an assignment.
- Here, the const version could just as well return a value, since the returned value is small.

Resource Management

Dynamic memory is an example of a resource that is allocated and must be released.

```
void f() {
    Point* p = new Point(10, 20);
    // ... use p
    delete p;
}
```

- If an exception is thrown while the object is used, `delete` will never be executed and the resource will never be released.
- You could try to catch all exceptions, delete the object and rethrow the exception, but this is a lot of work.
- Instead you should use a `unique_ptr` (see next slide).

Resource Management With a Unique Pointer

```
void f() {
    unique_ptr<Point> p(new Point(10, 20));
    // ... use p
}
```

- A `unique_ptr` is a wrapper around a pointer.
- It implements the access operators (`operator*` and `operator->`)
- It has a destructor that deletes the object. The destructor is executed when the pointer goes out of scope, even if an exception is thrown.

Implementing a Unique Pointer Class

There are many issues to consider when implementing a class like `unique_ptr`. Here we just show how to implement the member access operators `*` and `->` so that an object of the class can be used as a pointer (the class is a template so it can be used for arbitrary types):

```
template <typename T>
class UniquePointer {
public:
    UniquePointer(T* p) : handle(p) {}
    ~UniquePointer() { delete handle; }
    T& operator*() const { return *handle; }
    T* operator->() const { return handle; }
private:
    T* handle;
};
```

Copying and Moving Unique Pointers

Unique pointers cannot be copied — it would be disastrous if two unique pointers owned the same object. But unique pointers can be moved.

```
template <typename T>
class UniquePointer {
public:
    ...
    UniquePointer(const UniquePointer&) = delete;
    UniquePointer& operator=(const UniquePointer&) = delete;
    UniquePointer(const UniquePointer&& rhs) noexcept
        : handle(rhs.handle) {
        rhs.handle = nullptr;
    }
    UniquePointer& operator=(const UniquePointer&& rhs) noexcept {
        delete handle;
        handle = rhs.handle;
        rhs.handle = nullptr;
    }
    ...
};
```

Implementing a Shared Pointer Class

A shared pointer must keep track of how many pointers that reference the owned object. This number is called the *reference count* or use count. When the reference count becomes zero the owned object is deleted. The class interface is similar to the unique pointer's:

```
class SharedPointer {
public:
    SharedPointer(Point* p);
    ~SharedPointer();
    Point& operator*() const { return *handle; }
    Point* operator->() const { return handle; }
    SharedPointer(const SharedPointer&);
    SharedPointer& operator=(const SharedPointer&);
private:
    Point* handle;
    size_t* use; // pointer to reference count
};
```

Shared Pointer, Constructor and Destructor

The constructor creates the reference count and initializes it to 1:

```
SharedPointer(Point* p) : handle(p), use(new size_t(1)) {}
```

The destructor decrements the reference count and deletes the object and the reference counter if the reference count has reached zero:

```
~SharedPointer() {
    if (--*use == 0) {
        delete handle;
        delete use;
    }
}
```

Shared Pointer, Copy Control

The copy constructor makes a shallow copy and increments the reference count:

```
SharedPointer(const SharedPointer& sp)
        : handle(sp.handle), use(sp.use) {
    ++*use;
}
```

The assignment operator adjusts the counters:

```
SharedPointer& operator=(const SharedPointer& rhs) {
    ++*rhs.use;
    if (--*use == 0) {
        delete handle;
        delete use;
    }
    handle = rhs.handle;
    use = rhs.use;
    return *this;
}
```

Overloading Function Call

In the standard library, functions are often passed as arguments to algorithms. It is possible to pass a pointer to a function, but it is often better to pass a *function object*, i.e., an object of a class that overloads the function call operator, `operator()`. It is better because calls can be inlined. Example:

```
struct Sin {
    double operator()(double x) const {
        return std::sin(x);
    }
};

void f(Sin sin_obj) {
    double y = sin_obj(1.0); // calls Sin::operator()(double)
}
```

See next slide for another example.

Functions With State

With function objects, something which is used exactly as a function can have state that is saved between function calls.

```
class Accumulator {
public:
    Accumulator() : sum() {}
    double get_sum() const { return sum; }
    void operator()(double x) { sum += x; }
private:
    double sum;
};

double f(const vector<double>& v) {
    Accumulator accum;
    for (double x : v) {
        accum(x); // looks like a normal function call
    }
    return accum.get_sum();
}
```

Conversions to a Class Type

With the `Integer` class, you should be able to write code like this:

```
Integer a = 5;
Integer b = a + 6;
```

When `a + 6` is compiled, the compiler searches for a function `operator+(Integer,int)`. There is no such function, but there is a function `operator+(Integer,Integer)` and the single-argument constructor `Integer(int)` provides a conversion from `int` to `Integer`.

```
a + 6 // compiled into operator+(a, Integer(6))
```

In this case the conversion is cheap, but sometimes it is more efficient to provide special versions of the mixed-mode operators (but then you have to write a lot of functions).

Conversions from a Class Type

There is a problem with using the single-argument constructor for conversion. Example:

```
Integer a = 5;
cout << a + 6.9; // prints 11 instead of 11.9
```

As in the previous example, the expression is compiled into `operator+(a, Integer(6.9))`. The implicit conversion from `double` to `int` is used for the constructor argument. Instead, you can use a *conversion operator* to convert an `Integer` object to an `int`:

```
class Integer {
public:
    operator int() const { return value; }
};
```

Now, `a + 6.9` is compiled into `a.operator int() + 6.9`.

Type Conversions, Summary

Two kinds of conversion that you can supply:

- *to* a class type: are performed with single-argument constructors. They are forbidden if you make the constructor explicit.
- *from* a class type: are performed with conversion operators, member functions of the form `operator int() const`.
- In the new standard, the conversion operators can be made `explicit`. `C++11`

Think very carefully before providing class type conversions. The C++ conversion rules are complex enough as they are, without any added possibilities (read 14.9.1–3 in Lippman if you're interested). Never provide both single-argument constructors and conversion operators that convert from/to the same type.

Inheritance

C++ terminology: a superclass is a base class, a subclass is a derived class. Inheritance syntax:

```
class Vehicle { ... };
class Car : public Vehicle { ... };
```

The type rules are the same as in Java: a pointer (or reference) of a base class type may point (refer) to objects of its class and all derived classes. Differences between inheritance in Java and in C++:

- C++ by default uses static binding of method calls.
- C++ has `public`, `private`, and `protected` inheritance.
- C++ has multiple inheritance.

Dynamic Binding, Virtual Functions

In Java, the type of the *object* determines which implementation of a function that is called. This is called dynamic binding. To achieve this in C++ you must specify the function as `virtual`.

```
class A {
public:
    virtual void f() const { ... }
};

class B : public A {
public:
    virtual void f() const override { ... }
};

A* pa = new A; pa->f(); // calls A::f
pa = new B;    pa->f(); // calls B::f
```

Repeated `virtual`, `override`

- It is not necessary to repeat `virtual` on the overriding functions in derived classes.
- `override` C++11 checks that there really is a virtual function to override. This catches many errors (`f` and `g` in `B` do not override `f` and `g` in `A`, instead new functions are defined):

```
class A {
public:
    virtual void f() const;
    virtual void g(int x);
};

class B : public A {
public:
    virtual void f(); // not const
    virtual void g(double x); // different parameter type
};
```

Virtual Functions and References

Virtual functions also work as expected when called through a reference. Example (same classes as on slide 176):

```
void check(const A& r) { // r may refer to an A object
    r.f();               // or a B object
}

void g() {
    A a;
    B b;
    check(a); // will call A::f
    check(b); // will call B::f
}
```

Don't forget the `&` on the parameter — if you do, the parameter is called by value and the object `r` in `check` will always be an `A` object ("slicing").

Initializing Objects of Derived Classes

The constructor in a derived class must explicitly call the constructor in the base class (done with `super(...)` in Java). The call must be written in the constructor initialization list.

```
class Shape {
public:
    Shape(int ix, int iy) : x(ix), y(iy) {}
    ...
};

class Square : public Shape {
public:
    Square(int ix, int iy, int is) : Shape(ix, iy), s(is) {}
    ...
};
```

Pure Virtual (Abstract) Functions

An operation may be specified as abstract, “pure virtual” in C++, by adding `= 0` after the function signature:

```
class Shape {
public:
    virtual void draw(const Window& w) const = 0; // abstract
    ...
};

class Square : public Shape {
public:
    virtual void draw(const Window& w) const override { ... }
    ...
};
```

You cannot create an object of a class that contains pure virtual functions.

Interfaces

Java has interfaces, classes with only abstract operations and no attributes. C++ has no separate concept for interfaces; instead you use classes with only pure virtual functions and public inheritance (multiple inheritance, if a class implements more than one interface).

```
class Drawable { // interface
public:
    virtual ~Drawable() = default;
    virtual void draw(const Window& w) const = 0;
};

class Square : public Drawable { // "implements Drawable"
public:
    virtual void draw(const Window& w) const override { ... }
    ...
};
```

Type Rules, Runtime Type Information (RTTI)

A pointer (or reference) of a base class type may point (refer) to objects of its class and all derived classes.

```
Shape* psh = new Shape(1, 2);
Square* psq = new Square(2, 3, 5);

psh = psq; // always correct, all squares are shapes (upcast)
psq = psh; // not allowed, needs an explicit cast (downcast)
```

In Java, every object carries information about its type (used in method dispatch and in tests with `instanceof`). This type information is called runtime type identification, RTTI.

In C++, only objects of classes with at least one virtual function have RTTI. This is because it entails some memory overhead (one extra pointer in each object).

Downcasting Pointers and References

Downcasting is performed with `dynamic_cast`:

```
dynamic_cast<newtype*>(expression)
```

If the cast succeeds, the value is a pointer to the object. If it fails, i.e., if the expression doesn't point to an object of type `newtype`, the value is null.

```
Shape* psh = list.firstShape();
if (Square* psq = dynamic_cast<Square*>(psh)) {
    cout << "Side length: " << psq.getSide() << endl;
}
```

`dynamic_cast` can also be used with references. Since there are no "null references" the cast throws `std::bad_cast` if it fails, instead of returning a null value.

Copying Base Classes and Derived Classes

The rules for upcasting and downcasting concern pointers and references. There are similar rules for copying objects:

```
Shape sh(1, 2);
Square sq(2, 3, 5);

sh = sq; // allowed, but Square members are not copied (slicing)
sq = sh; // not allowed, would leave the Square members undefined
```

Note that slicing can occur if a function argument is passed by value:

```
void f(const Shape s) { // probably meant Shape&
    ...                 // here s is a Shape object,
    s.draw();           // Shape::draw is always called
}
```

Public and Private Inheritance

This is "normal" inheritance:

```
class A { ... };
class B : public A { ... };
```

The relationship `B` "is-an" `A` holds. Anywhere that an `A` object is expected, a `B` object can be supplied. In C++, there also is private (and protected) inheritance:

```
class A { ... };
class B : private A { ... };
```

`B` inherits everything from `A` but the public interface of `A` becomes private in `B`. A `B` object cannot be used in place of an `A` object. The relationship is that `B` "is-implemented-using" `A`.

Private Inheritance, Example

We wish to implement a class describing a stack of integers and decide to use the standard `vector` class to hold the numbers (in practice, we would have used the standard `stack` class).

We have several alternatives. This one isn't a serious candidate:

```
class Stack : public vector<int> { // public inheritance
public:
    void push(int x) { push_back(x); }
    int pop() { int x = back(); pop_back(); return x; }
};
```

Not good: we get the functionality we need, but also much more. The class also inherits, among others, the `insert` function which can insert an element at any location in the vector.

Private Inheritance Example, cont'd

```
class Stack { // composition, like in Java
public:
    void push(int x) { v.push_back(x); }
    int pop() { int x = v.back(); v.pop_back(); return x; }
private:
    vector<int> v;
};
```

Good: only push and pop are visible. But note that the operations must delegate the work to the `vector` object, which means some overhead.

```
class Stack : private vector<int> { // private inheritance
public:
    void push(int x) { push_back(x); }
    int pop() { int x = back(); pop_back(); return x; }
};
```

Good: only push and pop are visible, no overhead.

Exposing Individual Members

We wish to add a `bool empty()` member to the `Stack` class. It can be done like this:

```
class Stack : private vector<int> {
public:
    bool empty() const { return vector<int>::empty(); }
    ...
};
```

But it's better to expose the member `empty()` from `vector<int>`:

```
class Stack : private vector<int> {
public:
    using vector<int>::empty;
    ...
};
```

Copy Control

If a base class defines a copy constructor and an assignment operator, they must be called from the derived classes:

```
class A {
public:
    A(const A& a) { ... }
    A& operator=(const A& rhs) { ... }
};

class B : public A {
public:
    B(const B& b) : A(b) { ... }
    B& operator=(const B& rhs) {
        if (&rhs == this) {
            return *this;
        }
        A::operator=(rhs);
        ...
    }
};
```

Destructing Derived Objects

Suppose that classes A and B manage separate resources. Then, each class needs a destructor:

```
class A {
public:
    ~A();
    ...
};

class B : public A {
public:
    ~B();
};
```

When a B object is destroyed and the destructor ~B() is called, the destructor for the base class, ~A(), is automatically called. This continues up to the root of the inheritance tree.

Virtual Destructors

There is a problem with destructors in derived objects. Consider the following statements:

```
A* pa = new B;
...
delete pa;
```

The wrong destructor, `~A()`, will be called when `pa` is deleted, since also destructors use static binding by default. To correct this error, you must make the destructor `virtual`:

```
virtual ~A();
```

Most base classes should have virtual destructors.

Multiple Inheritance

The relationship “is-a” can hold between a class and more than one other class. A panda is a bear and an endangered species, a teaching assistant is a student and a teacher, an amphibious vehicle is a land vehicle and a water vehicle, . . .

In C++, a class may be derived from several base classes:

```
class TeachingAssistant : public Student, public Teacher {
    ...
};
```

Think carefully before you use multiple inheritance. There are problems with data duplication that must be considered. Multiple inheritance of interfaces (classes with only pure virtual functions) doesn’t give any problems.

Conversions and Member Access

Since public inheritance is used, the normal rules for conversion to a base class apply:

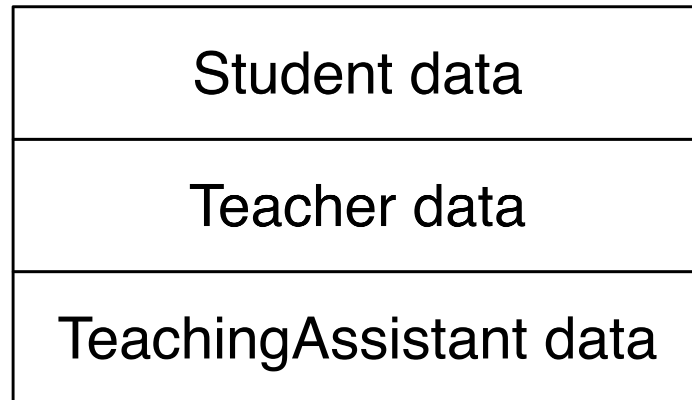
```
void f(const Student& s) { ... }
void g(const Teacher& t) { ... }

TeachingAssistant ta1;
f(ta1); // ok, ta1 is a student
g(ta1); // ok, ta1 is a teacher
```

A derived class sees the union of the members of the base classes. Name conflicts must be explicitly resolved.

```
cout << ta1.getProgram() << endl; // from Student
cout << ta1.getSalary() << endl;  // from Teacher
ta1.Student::print(); // if both Student and Teacher
ta1.Teacher::print(); // have a print() member
```

Memory Layout

A `TeachingAssistant` object looks like this in memory:



There is an obvious problem: what about duplicated data? If, for example, both students and teachers have names, a teaching assistant will have two names. This can be avoided using *virtual* inheritance.

Replicated Base Classes

We start by factoring out the common data (name, address, ...) and place it in a separate class `Person`:

```
class Person { ... };
class Student : public Person { ... };
class Teacher : public Person { ... };
class TeachingAssistant : public Student, public Teacher { ... };
```

However, this doesn't solve the problem. A `TeachingAssistant` object will contain all `Student` data, which contains `Person` data, and `Teacher` data, which also contains `Person` data.

Virtual Inheritance

The duplication of data can be avoided by specifying the inheritance of the common base class as `virtual`:

```
class Person { ... };
class Student : public virtual Person { ... };
class Teacher : public virtual Person { ... };
class TeachingAssistant : public Student, public Teacher { ... };
```

When virtual inheritance is used, data from a common base class is only stored once in an object.

We have only scratched on the surface of multiple inheritance. There is much more to consider: initialization order, destruction order, virtual functions, conversions to virtual base classes, ...

Templates

A class that describes a stack of integers is easy to write. If you need a class of doubles you must write another, very similar, class. With a *class template* you can write a stack where the elements are of an arbitrary type — you supply the actual type as an argument when you *instantiate* the template.

The library containers are class templates: you can create a `vector<int>`, `vector<double>`, `vector<Point>`, ...

There are also *function templates*. You can write a function that takes arguments of arbitrary types.

Note: Lippman mixes the presentation of class templates and function templates. Here, we first present function templates and then class templates.

Defining a Function Template

This is a function template that compares two values of the same type:

```
template <typename T>
int compare(const T& v1, const T& v2) {
    if (v1 < v2) { return -1; }
    if (v2 < v1) { return 1; }
    return 0;
}
```

- `T` is a template parameter, which must be a type.
- You may write `class` instead of `typename` — use `typename` in new programs.

Using a Function Template

When you call a template function, the compiler deduces what types to use instead of the template parameters and instantiates ("writes") a function with the correct types.

```
void f() {
    cout << compare(1, 0) << endl;
    // T is int, the compiler instantiates
    //      int compare(const int&, const int&)

    string s1 = "hello";
    string s2 = "world";
    cout << compare(s1, s2) << endl;
    // T is string, the compiler instantiates
    //      int compare(const string&, const string&)
}
```

Requirements on Template Parameter Types

A template usually puts some requirements on the argument types. If these requirements are not met the instantiation will fail. Example:

```
void f() {
    Point p1(10, 20);
    Point p2(10, 30);
    cout << compare(p1, p2) << endl;
    // T is Point, the compiler tries to instantiate
    //     int compare(const Point&, const Point&),
    // this doesn't compile since Point objects
    // cannot be compared with <
}
```

- If class `Point` implements `operator<`, everything is ok.
- You should try to keep the number of requirements on argument types as small as possible (see next slide).

Writing Type-Independent Code

The only requirement placed on the type `T` in the `compare` template is that objects of `T` can be compared with <. The following implementation puts more requirements on `T` and isn't good:

```
template <typename T>
int compare(T v1, T v2) {
    if (v1 < v2) { return -1; }
    if (v1 == v2) { return 0; }
    return 1;
}
```

- The arguments are passed by value, so it must be possible to copy objects of `T`.
- Objects of `T` must implement comparison with < *and* ==.

Concepts and Models

A *concept* is a set of requirements that a template argument must meet so that the template can compile and execute properly.

Requirements are specified as generally as possible: instead of saying that "class `T` must define the member function `operator++`", you say "for any object `t` of type `T`, the expression `++t` is defined". (It is left unspecified whether the operator is a member or a global function and whether `T` is a built-in type or a user-defined type.)

An entity that meets the requirements of a concept is a *model* of the concept. For example, the class `string` is a model of the concept `LessThanComparable`, which requires that objects can be compared with <.

Basic Concepts

Some important concepts:

DefaultConstructible Objects of type `X` can be constructed without initializing them to any specific value.

Assignable Objects of type `X` can be copied and assigned.

LessThanComparable Objects of type `X` are totally ordered (`x < y` is defined).

EqualityComparable Objects of type `X` can be compared for equality (`x == y` and `x != y` are defined).

All built-in types are models of these concepts. Note that concepts are just words that have no meaning in a program. A proposal to incorporate concepts in the language was rejected at a late stage in the C++11 standardization process, but work continues on a modified proposal.

Type Parameters Must Match

During template instantiation the argument types must match exactly — no conversions are performed. The following call to `compare` will fail:

```
int i = ...;
double d = ...;
cout << compare(i, d) << endl;
```

With this version of the template, the instantiation will succeed:

```
template <typename T, typename U>
int compare2(const T& v1, const U& v2) {
    if (v1 < v2) { return -1; }
    if (v2 < v1) { return 1; }
    return 0;
}
```

A < operator must exist that can compare a `T` object and a `U` object.

Explicit Instantiation

If a function template instantiation fails because the types of the arguments don't match, you can make an explicit instantiation of the template:

```
int i = ...;
double d = ...;
cout << compare<double>(i, d) << endl;
```

Here, `i` is widened to `double` in accordance with the usual conversion rules.

Return Types

A function to compute the smallest of two values of different types:

```
template <typename T, typename U>
T min(const T& v1, const U& v2) {
    return (v1 < v2) ? v1 : v2;
}
```

This is not correct: the return value always has the first argument's type, so `min(4, 2.3)` becomes 2. The return type should be "the most general of the types `T` and `U`", i.e. the type to which the compiler will convert an expression containing `T` and `U objects`.

This can be expressed in the following way `C++11`:

```
template <typename T, typename U>
auto min(const T& v1, const U& v2) -> decltype(v1 + v2) {
    return (v1 < v2) ? v1 : v2;
}
```

Template Compilation

When an “ordinary” function is called, the compiler needs to see only the function declaration. For an inline function, code will be generated at the call site, so the function *definition* must be available at the call.

This means that inline function definitions should be placed in header files. The same holds for template function definitions.

The most common errors when templates are used are related to types: trying to instantiate a template with arguments of types that don’t support all requirements. Such errors are reported during instantiation of the template.

Class Templates

A stack template:

```
template <typename T>
  // T must be DefaultConstructible (new T[size]) and
  // Assignable (v[top++] = item, return v[--top])
class Stack {
public:
    Stack(size_t size) : v(new T[size]), top(0) {}
    ~Stack() { delete[] v; }
    void push(const T& item) { v[top++] = item; }
    T pop() { return v[--top]; }
    bool empty() const { return top == 0; }
    Stack(const Stack&) = delete;
    Stack& operator=(const Stack&) = delete;
private:
    T* v;
    size_t top;
};
```

Defining Template Class Members

It is possible to write the definition of a class member function outside the class definition, but then the template information must be repeated:

```
template <typename T> // this goes in a header file
class Stack {
    ...
    void push(const T& item);
    ...
};

template <typename T> // this goes in the same header file
void Stack<T>::push(const T& item) { v[top++] = item; }
```

Using a Class Template

With function templates, the compiler can deduce template parameter types from the function calls. With class templates, the types must be explicitly supplied when an object is created.

```
Stack<char> cs(100);
Stack<Point> ps(200);

cs.push('X');
ps.push(Point(10, 20));
...
char ch = cs.pop();
```

Nontype Template Parameters

A template parameter need not be a type, it can also be a constant value. And template parameters may have default values:

```
template <typename T = int, size_t size = 100>
class Stack {
public:
    Stack() : top(0) {}
    ...
private:
    T v[size];
    size_t top;
};

void f() {
    Stack<double, 200> s1; // 200 doubles
    Stack<Point> s2;       // 100 Points
    Stack s3;              // 100 ints
    ...
}
```

Types Inside the Template Definition

Template classes often export type aliases:

```
template <typename T>
class vector {
public:
    using value_type = T;
    ...
};
```

This means that you can define variables of the vector's element type:

```
void sort(vector<int>& v) {
    ...
    vector<int>::value_type tmp = v[i];
    ...
}
```

Here, you could just as well have written `int` — but see next slide.

Using Exported Types in a Template, typename

Now suppose that the function `sort` is a template that can sort both vectors and other containers (of type `T`). The element type is `T::value_type`, but before the type specifier you must write `typename`. This is because the compiler cannot determine whether `T::value_type` is a type or a (static) member variable.

```
template <typename T>
void sort(T& v) {
    ...
    typename T::value_type tmp = v[i]; // C++11: auto tmp =
    ...
}
```

Rule: write `typename` before the type specifier when using a type that depends on a template parameter.

Template Metaprogramming

A template can instantiate itself recursively, so the "template language" is in fact a Turing-complete programming language. Suppose that you in a program need values of $n!$, where the n-s are constant:

```
template <int n> struct Factorial {
    static const int value = n * Factorial<n - 1>::value;
};

template <> struct Factorial<1> { // specialization for n = 1
    static const int value = 1;
};

int main() {
    cout << Factorial<6>::value << endl;
}
```

The value 720 is computed by the *compiler* through recursive instantiations of the class.

About the Book and the Slides

We return to the standard library, chapters 9–11 of Lippman. However, we will treat the material in almost the reverse order from Lippman:

- **Iterators** 3.4, 9.2.1–9.2.3, 10.4
- **Function objects** 10.3
- **Algorithms** 10.1–10.2, 10.5–10.6
- **Containers** Chapters 9 and 11

Containers, Algorithms, Iterators

The standard library (also called the standard template library, STL) provides these components:

- **Containers** for homogenous collections of values (vectors, deques, lists, sets, maps, stacks, queues, priority queues).
- **Algorithms** for operating on containers (searching, sorting, copying, ...).
- **Iterators** for iterating over containers.

One important design goal for the library was efficiency. Since everything builds on templates, there is no execution-time penalty for using the library.

The library is *not* an object-oriented library, although inheritance is used internally in the implementation. There is nothing like Java's strange `Collection` hierarchy.

Iterators

We have already used iterators to traverse `vector`'s. You can:

- set an iterator to the start of a container,
- access the value that an iterator "points to",
- increment the iterator to point to the next value,
- check if the iterator has reached the end of a container.

A pointer is an iterator for arrays. We have done things like the following:

```
int ia[] = {5, 7, 2, 3};
for (int* p = ia; p != ia + 4; ++p) {
    cout << *p << endl;
}
```

An Algorithm Using Pointers

The following function finds the first occurrence of a value in an `int` array, from the address `beg` up to, but not including, `end`:

```
int* find(const int* beg, const int* end, int value) {
    while (beg != end && *beg != value) {
        ++beg;
    }
    return beg;
}
```

This should be possible to generalize:

- The function applies only to arrays of `int`'s.
- The algorithm (linear search) is usable for any linear sequence (array, vector, linked list, ...), but as it's written the function applies only to arrays.

A Generic Algorithm

With a function template you can write a generic algorithm:

```
template <typename InputIterator, typename T>
InputIterator
find (InputIterator beg, InputIterator end, const T& value) {
    while (beg != end && *beg != value) {
        ++beg;
    }
    return beg;
}
```

This version of `find` works equally well as the previous one for `int` arrays:

```
int ia[] = ...;
int nbr = ...;
int* p = find(ia, ia + 4, nbr);
if (p != ia + 4) {
    cout << nbr << " found at pos " << p - ia << endl;
} else {
    cout << nbr << " not found" << endl;
}
```

Using the Generic Algorithm

The generic algorithm can be used for arrays of all types (that support equality checking with `!=`):

```
double id[] = ...;
double nbr = ...;
double* p = find(id, id + 4, nbr);
```

And it can be used for other containers (that have iterators):

```
vector<int> v = ...;
int nbr = ...;
vector<int>::iterator p = find(v.begin(), v.end(), nbr);
```

The `find` algorithm is one of the standard algorithms in the library.

Requirements on Iterators

The find algorithm places the following requirements on an `InputIterator`:

- must be `Assignable` (argument passed by value).
- must be `EqualityComparable` (`!=`).
- must support dereferencing for reading (`*`).
- must support prefix increment (`++`).

Iterators for other containers must meet these requirements. See the following slides for an example.

Given a container, it must also be possible to find the begin and end of the container (like the `begin()` and `end()` functions in `vector`).

A Linked List (Sketch Only)

In a singly-linked list of `int`'s, a list and a list node could look like this (many details are omitted):

```
class List {
public:
    using iterator = ListIterator;
    iterator begin() { return iterator(first); }
    iterator end() { return iterator(nullptr); }
    ...
private:
    Node* first;
};

struct Node {
    int data;
    Node* next;
};
```

An Iterator Class (Sketch Only)

The `ListIterator` class can look like this:

```
struct ListIterator {
    Node* current;

    explicit ListIterator(Node* c) : current(c) {}

    bool operator!=(const ListIterator& rhs) const {
        return current != rhs.current;
    }

    int& operator*() { return current->data; }

    ListIterator& operator++() {
        current = current->next;
        return *this;
    }
};
```

Getting Iterators From Containers

Each library container has two functions `begin()` and `end()`. `begin()` returns an iterator to the first element of the container, `end()` returns an iterator one past the last element.

```
template <typename T>
class vector {
public:
    using iterator = T*;
    iterator begin() { return values; }
    iterator end() { return values + size; }
    ...
private:
    T* values;   // dynamically allocated array of T's
    size_t size; // number of elements
};
```

This is an example only — it is not guaranteed that a `vector` iterator is a pointer.

Versions of `begin()` and `end()`

Each library container class defines two iterator types:

`iterator` Allows reading and writing (`x = *it` and `*it = x`)

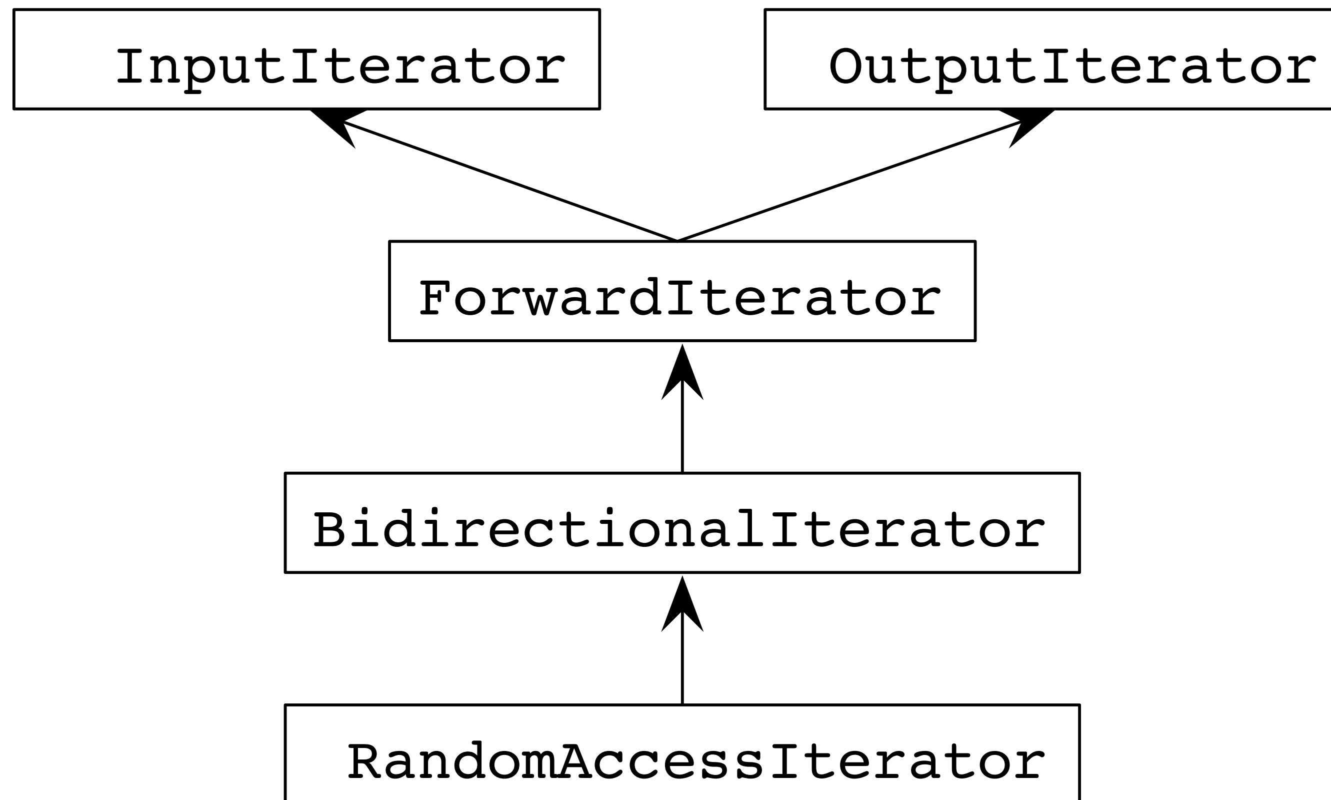
`const_iterator` Only allows reading (`x = *cit`)

The `begin()` and `end()` functions are overloaded on `const`. They return const iterators for const objects, non-const iterators for non-const objects. The new library introduced `cbegin()` and `cend()` members that always return const iterators `C++11`. Examples:

```
void f(const vector<int>& v1) {
    vector<int> v2;
    auto it1 = v1.begin();  // vector<int>::const_iterator
    auto it2 = v2.begin();  // vector<int>::iterator
    auto it3 = v1.cbegin(); // vector<int>::const_iterator
    auto it4 = v2.cbegin(); // vector<int>::const_iterator
}
```

Iterator Concepts

The requirements on an input iterator are weak; often more “powerful” iterators are needed. There is a hierarchy among iterator concepts:



Iterator Concepts, Details

An iterator "points to" a value. All iterators are `DefaultConstructible` and `Assignable` and support `++it` and `it++`. Additional concepts:

- `InputIterator` Can be dereferenced to read a value, is `EqualityComparable`.
- `OutputIterator` Can be dereferenced to (over)write a value.
- `ForwardIterator` Can both read and write.
- `BidirectionalIterator` Can move both forwards and backwards.
- `RandomAccessIterator` Allows pointer arithmetic and subscripting.

A built-in pointer is a model of `RandomAccessIterator`.

Input and Output Iterators, Example

The following algorithm copies the range [beg, end) (from beg up to but not including end) to the range starting at dest.

```
template <typename InIt, typename OutIt>
  // InIt is a model of InputIterator
  // OutIt is a model of OutputIterator
OutIt copy(InIt beg, InIt end, OutIt dest) {
    for (; beg != end; ++beg, ++dest) {
        *dest = *beg;
    }
    return dest;
}
```

- The implementation relies on the fact that the iterators are passed by value. This is normally the case — iterators are built-in pointers or "small" objects, so there is no efficiency penalty.
- If the iterators had been passed by constant reference you would have needed temporary variables in the algorithm.

Using copy

copy is one of the standard algorithms. It can be used like this:

```
void f() {
    int x[] = {1, 2, 3, 4};
    int y[10];
    int* last = copy(x, x + 4, y);

    vector<int> v = {5, 6, 7, 8, 9, 10};
    copy(v.begin(), v.end(), last);
}
```

- Note that there must be enough memory allocated for the destination range — the copy algorithm cannot allocate any memory (it doesn't know how or where). Examples see next slide.

Iterators Don't Allocate Memory

The following uses of `copy` are wrong:

```
vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int y[5];
copy(v.begin(), v.end(), y); // y is too short

vector<int> destV;
copy(v.begin(), v.end(), destV.begin()); // destV has no elements
```

- The first error must be fixed by creating a longer array.
- In the second call to `copy` all would be ok if the destination iterator did `destV.push_back` on the values instead of just writing. For this, the iterator must know which vector it's supposed to `push_back` on.

Insert Iterators

Iterators that know about their container and know how to insert a value are provided by the library in the form of iterator adapters called *insert iterators* or inserters. There are three kinds:

Back inserters `*it = value`: `operator=` calls `push_back(value)`, `operator*` and `operator++` do nothing.

Front inserters same, but calls `push_front(value)` (insert at front).

General inserters same, but calls `insert(pos, value)` (insert at position `pos`).

The copying problem can be solved with a back inserter:

```
vector<int> destV;
copy(v.begin(), v.end(), back_inserter(destV));
```

Stream Iterators, Input

The iterator adapter `istream_iterator` functions as an input iterator but the values are read from a stream. It takes a stream as an argument to the constructor and the value type as a template argument. The default constructor creates an iterator that represents the end of a stream. Example:

```
istream_iterator<string> input(cin); // read strings from cin
istream_iterator<string> end;        // end iterator
vector<string> v;
while (input != end) {
    v.push_back(*input++);
}
```

Better, using the copy algorithm:

```
copy(input, end, back_inserter(v));
```

Counting Words in a String

This was an exam question: count the number of words in a string `s`. Words are separated by whitespace.

- Standard solution:

```
istringstream iss(s);
string temp;
int words = 0;
while (iss >> temp) {
    ++words;
}
```

- More elegant solution — the library function `distance` computes the distance between two iterators:

```
istringstream iss(s);
int words = distance(istream_iterator<string>(iss),
                     istream_iterator<string>());
```

Stream Iterators, Output

The iterator adapter `ostream_iterator` functions as an output iterator but the values are written to a stream. It takes a stream as an argument to the constructor and the value type as a template argument. Optionally, a C-style string that will be written between the values can be specified as a second argument to the constructor.

```
vector<string> v;
...
copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
```

To write a newline after each value you must use the delimiter `"\n"`, the linefeed character. You cannot use `endl` (`endl` isn't a character or a string, but an I/O manipulator).

Reverse Iterators

A reverse iterator is an iterator that traverses a container backward. `++` on a reverse iterator accesses the previous element, `--` accesses the next element. The containers have types `reverse_iterator` and `const_reverse_iterator` and functions `rbegin()` and `rend()` (and crbegin() and crend() `C++11`).

Example (read words, print backwards):

```
vector<string> v;
copy(istream_iterator<string>(cin), istream_iterator<string>(),
     back_inserter(v));
copy(v.rbegin(), v.rend(),
     ostream_iterator<string>(cout, "\n"));
```

Iterator Traits

Sometimes, you must know more about an iterator than just the fact that it is an iterator. Suppose that you wish to write a function to sort the values in an iterator range, and that you need to define a temporary variable to swap two values:

```
template <typename It>
void sort(It beg, It end) { ... elem_type temp = *beg; ... }
```

The type of the value to which an iterator points is available from the class `iterator_traits`:

```
using elem_type = typename iterator_traits<It>::value_type;
```

Use `auto` in C++11.

Iterator Notes

- For efficiency reasons, use `++x` instead of `x++` when incrementing an iterator (`--x` when decrementing a bidirectional or random-access iterator).
- You must know what kind of iterator you are dealing with. For instance, this is allowed only for random-access iterators:

```
Iterator first = ...;
Iterator next = first + 1;
```

  But `Iterator next = first; ++next;` is legal for all iterators.
- The first expression below isn't legal for any iterator, but the equivalent second expression is legal for random-access iterators:

```
Iterator mid = (beg + end) / 2;
Iterator mid = beg + (end - beg) / 2;
```

Functions and Function Objects

Regular functions and function objects (objects of a class that overloads the function call operator) are equivalent — they are called in the same way and may both be passed as arguments to other functions. Function objects (sometimes called functors) may have state and are often more efficient to use than regular functions. This is because calls of a function object can be inlined.

Algorithms and Functions

Functions and function objects are used to parameterize an algorithm with an “operation”. The library function `for_each` traverses a range and performs an operation on all elements:

```
template <typename It, typename Op>
  // It is an InputIterator
  // Op is a function with one parameter
Op for_each(It beg, It end, Op op) {
    for (; beg != end; ++beg) {
        op(*beg);
    }
    return op;
}
```

The argument op may be a function or a function object, as long as it has one parameter.

for_each, Regular Functions

for_each can be used with regular functions:

```
void clear(int& value) { value = 0; }

void print(int value) { cout << value << " "; }

int main() {
    vector<int> v = {10, 20, 30, 40, 50};

    for_each(v.begin(), v.end(), print);
    cout << endl;

    for_each(v.begin(), v.end(), clear);

    for_each(v.begin(), v.end(), print);
    cout << endl;
}
```

for_each, Function Objects

On slide 171 we showed an example of a function object with state. The same class, now as a class template, and an example of use:

```
template <typename T>
class Accumulator {
public:
    Accumulator() : sum() {}
    T get_sum() const { return sum; }
    void operator()(const T& x) { sum += x; }
private:
    T sum;
};

vector<int> v = {10, 20, 30, 40, 50};
Accumulator<int> accum;
accum = for_each(v.begin(), v.end(), accum);
cout << accum.get_sum() << endl;
```

Function Examples — Finding

The library algorithm `find_if` returns an iterator to the first element for which the function `pred` returns true.

```
template <typename InIt, typename Pred>
InIt find_if(InIt beg, InIt end, Pred pred) {
    while (beg != end && !pred(*beg)) {
        ++beg;
    }
    return beg;
}
```

We will use `find_if` to find an element smaller than a given value:

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(), ???) != v.end();
}
```

Finding With a Regular Function

A regular function can be used, but the searched-for value must be stored in a global variable. This is not an attractive solution.

```
int global_value;

bool is_less(int x) { return x < global_value; }

bool exists_smaller(const vector<int>& v, int value) {
    global_value = value;
    return find_if(v.begin(), v.end(), is_less) != v.end();
}
```

Finding With a Function Object

A solution with a function object is much more flexible:

```
class Less {
public:
    Less(int v) : value(v) {}
    bool operator()(int x) const { return x < value; }
private:
    int value;
};

bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(), Less(value)) != v.end();
}
```

Also, this solution is more efficient than the previous one. The call to the comparison function can be inlined by the compiler. This is not possible when the argument is a function pointer.

Finding With a Lambda C++11

In the new standard, the definition of a function object can be written at the place of the call. This is called a lambda function.

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(),
                [value](int x) { return x < value; }) != v.end();
}
```

- `[value]` access the variable `value` in the current scope. Variables can also be captured by reference: `[&value]`
- `(int x)` normal parameter list
- `) {` the return type is deduced from the `return` statement. Can be specified as `-> bool`
- `{ ... }` function body

The Compiler Generates a Class From a Lambda

When you define a lambda, the compiler writes a function class and creates a function object:

```
auto less = [value](int x) -> bool { return x < value; }
```

This is what the compiler generates (compare with slide 244):

```
class Less {
public:
    Less(int v) : value(v) {}
    bool operator()(int x) const { return x < value; }
private:
    int value;
};

auto less = Less(value);
```

Library-Defined Function Objects

The algorithm `sort(beg,end)` sorts an iterator range in ascending order (values are compared with <). A function defining the sort order can be supplied as a third argument. To sort in descending order (comparing values with >):

```
vector<string> v;
...
sort(v.begin(), v.end(),
        [](const string &a, const string& b) { return a > b; });
```

Simple function objects like `greater`, `less`, `equal_to` are available in the library. They can be used like this:

```
sort(v.begin(), v.end(), greater<string>());
```

Binding Function Arguments C++11

The library-defined function objects can often be used together with the library algorithms, but sometimes they don't exactly match. We return to the problem of finding an element less than a given value (slide 242).

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(), ???) != v.end();
}
```

`find_if` takes each element from the vector and checks if the function returns true. Here, each element should be checked against `value`. The library function object `less` takes two parameters, and we must *bind* the second argument to `value`:

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(),
                   bind(less<int>(), _1, value)) != v.end();
}
```

Algorithms

There are about 100 algorithms in the header `<algorithm>`. They can be classified as follows:

Nonmodifying look at the input but don't change the values or the order between elements (`find`, `find_if`, `count`, ...).

Modifying change values, or create copies of elements with changed values (`for_each`, `copy`, `fill`, ...).

Removing remove elements (`remove`, `unique`, ...).

Mutating change the order between elements but not their values (`reverse`, `random_shuffle`, ...).

Sorting a special kind of mutating algorithms (`sort`, ...).

Sorted Range require that the input is sorted (`binary_search`, ...).

Numeric for numeric processing (`accumulate`, `inner_product`, ...).

Algorithm Notes

- The input to an algorithm is one or more iterator ranges `[beg,end)`. If the algorithms writes elements, the start of the output range is given as an iterator.
- Return values are often iterators.
- Some algorithms are used for the same purpose and have the same number of arguments. They are not overloaded, instead they have different names. An example is `find` for finding an element with a given value, `find_if` for finding an element for which a predicate is true.
- Some containers have specialized algorithms that are more efficient than the library algorithms, for instance `map::find` for finding a value in a map (binary search tree).

Finding Algorithms

We have already seen examples of algorithms that find values. These algorithms return the end iterator if the value isn't found. Other examples:

```
vector<string> v = ...;

cout << "There are " << count(v.begin(), v.end(), "hello")
     << " hello's in v" << endl;

vector<string> v2 = {"hello", "hej", "davs"};
auto it = find_first_of(v.begin(), v.end(), v2.begin(), v2.end());
```

Modifying Algorithms

The following examples use strings — strings have iterators that can be used with the library algorithms.

```
string s = "This is a sentence";

// convert the string to lower case. The C function
// tolower(ch) returns ch in lower case
transform(s.begin(), s.end(), s.begin(), ::tolower);

// print all non-blanks
copy_if(s.begin(), s.end(), ostream_iterator<char>(cout),
        [](char c) { return c != ' '; });

// fill s with blanks
fill(s.begin(), s.end(), ' ');
```

Removing Algorithms

The most important fact about the removing algorithms is that they don't actually remove elements from a container (since the algorithms have iterators as input they don't know which container to remove from, or how). Instead they reorder the elements based on a criterion.

```
vector<int> v = {1, 3, 2, 6, 3, 3, 4};

auto it = remove(v.begin(), v.end(), 3);
// v contains {1, 2, 6, 4, ...}, it points after the 4
// vector::erase really removes elements:
v.erase(it, v.end());

v = {1, 3, 3, 5, 7, 7, 7, 8, 9, 9};
v.erase(unique(v.begin(), v.end()), v.end());
```

Mutating and Sorting Algorithms

```
vector<int> v = {1, 4, 44, 3, 15, 6, 5, 7, 7, 4, 22, 1};
// reorder so odd numbers precede even numbers
partition(v.begin(), v.end(), [](int x) { return x % 2 != 0; });

// shuffle the elements (requires random access iterators)
random_shuffle(v.begin(), v.end());

// sort until the first 5 elements are correct
partial_sort(v.begin(), v.begin() + 5, v.end());
```

Sorted Range Algorithms

These algorithms require that the input range is sorted. Example (insert unique numbers in a sorted vector):

```
int main() {
    vector<int> v;
    int nbr;
    while (cin >> nbr) {
        auto it = lower_bound(v.begin(), v.end(), nbr);
        if (it == v.end() || *it != nbr) {
            v.insert(it, nbr);
        }
    }
    ...
}
```

Algorithm Example

Read words from standard input, remove duplicate words, print in sorted order:

```
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>

int main() {
    using namespace std;
    vector<string> v((istream_iterator<string>(cin)),
                     istream_iterator<string>());
    sort(v.begin(), v.end());
    unique_copy(v.begin(), v.end(),
                ostream_iterator<string>(cout, "\n"));
}
```

Algorithm Summary

- Every C++ programmer should be familiar with the algorithms.
- The algorithms are easy to use, well tested and efficient.
- We have only shown a few of the algorithms. Look in a book or at documentation on the web, for example at
    - `http://en.cppreference.com/w/cpp/algorithm`, or
    - `http://www.cplusplus.com/reference/algorithm`

Different Kinds of Containers

A container is an object that stores other objects and has methods for accessing the elements. There are two fundamentally different kinds of containers:

Sequences Elements are arranged in linear order. The sequences are `vector`, `list`, and `deque`. New in C++11 are `array` and `forward_list` [C++11]. The library class `string` supports most sequence features (for example it has iterators), but it is usually not used as a container.

Associative Containers Elements are arranged in an order defined by the container. The associative containers are `set`, `multiset`, `map`, and `multimap` (implemented using search trees). New in C++11 are `unordered_` versions of these containers (implemented using hash tables) [C++11].

Container Notes

- Every container type has associated iterator types and functions that return iterators.
- Containers use value semantics. When an element is inserted into the container, its value is copied. When an element is removed, its destructor is called. When a container is destroyed, all its elements are destroyed.
- You usually don't store raw pointers in containers, since then you must delete the element when it's removed from the container. Use safe pointers if you have to store pointers (when you wish to store objects in an inheritance hierarchy, when the objects are very expensive to copy, . . .).

C++11 Container Notes C++11

- Containers have move constructors (and move assignment operators). This means, among other things, that it is efficient to return even a large container by value from a function:

```
vector<string> f() {
    vector<string> local;
    ...
    return local;
}
```

- In addition to the functions that insert elements by copying (or moving) them, elements can be constructed in place:

```
vector<Person> v;
Person p("Bob", "Berlin");
v.push_back(p);                        // copy
v.push_back(Person("Alice", "Lund")); // move
v.emplace_back("Joe", "London");       // construct in place
```

Types Defined by Containers

Each container class defines a number of types that are associated with the container. Commonly used are:

`const_iterator` A const iterator (may be used to examine, but not to modify, elements).

`iterator` A non-const iterator.

`value_type` The type of the elements stored in the container.

`size_type` A type large enough to represent the size of the container (usually `unsigned long`, the same as `std::size_t`).

Operations on Containers

Operations that are defined for all containers :

- `X()` Create an empty container.
- `begin()` Iterator pointing to the first element, also `cbegin()`.
- `end()` Iterator pointing one past the last element, also `cend()`.
- `size()` Number of elements.
- `empty()` True if `size() == 0` (evaluated in constant time).

For almost all containers :

- `insert(p,t)` Insert `t` before `p` (not for `array` and `forward_list`).
- `clear()` Remove all elements (not for `array`).

Fixed-Size Arrays C++11

The new standard introduced the library class `array`, which is a wrapper around a built-in array. It has iterators like a vector and operations for element access. Example:

```
array<int, 10> v; // the size is part of the type, elements
                  // are undefined (like a built-in array)
fill(v.begin(), v.end(), 0);
...
int i = find_if(v.begin(), v.end(), [](int x) { return x != 0; })
        - v.begin();
```

Keep in mind that arrays are sequences but fixed size; there are no "dynamic" operations like `push_back`, `pop_back`, `clear`, ... Arrays are not treated in the rest of the slides.

Singly-Linked List C++11

The new standard introduced the library class `forward_list`, which is a singly-linked list. The operations reflect the fact that the list is singly linked (only forward iterators, `push_front` but no `push_back`, `insert_after` but no `insert`, ...

Compared to a `list` (which is doubly-linked), a node in a `forward_list` saves space for one pointer. Use a `forward_list` only if you need a linked list and memory is very tight. Forward lists are not treated in the rest of the slides.

Sequences

In addition to the common container operations, sequences have the following operations:

- `X(n)` Create a container with `n` default-constructed elements.
- `X(n,t)` Create a container with `n` copies of `t`.
- `front()` A reference to the first element.
- `back()` A reference to the last element.
- `push_back(t)` Append a copy of `t`.
- `pop_back()` Remove (but don't return) the last element.
- `erase(p)` Remove the element pointed to by p.
- `resize(n)` Resize the sequence to size `n`, inserting default-constructed elements at the end or erasing elements at the end.

Sequence Notes

The reason that there are different kinds of sequences is efficiency considerations:

`vector` Insertion and deletion at the end is fast, allows random access (has `operator[]` and random access iterators).

`deque` Insertion and deletion at the end *and* at the front is fast, allows random access.

`list` Insertion and deletion anywhere is fast, but you have no random access to elements and you pay a storage overhead since each list node has pointers to the next and previous node.

Use a `vector` unless you have very special reasons to use one of the other sequences. Bjarne Stroustrup has demonstrated that a vector nearly always wins over a list, even in applications where elements are inserted and removed in the middle.

Vectors

- A vector grows dynamically to accommodate new elements.
- A vector has a size (the number of elements currently in the vector) and a capacity (the maximum size to which the vector can grow before new storage must be allocated).
- A vector's capacity can be increased (storage can be pre-allocated) with a call to `reserve(n)`.

```
vector<int> v; // this works well
for (size_t i = 0; i != 1000000; ++i) { v.push_back(i); }

vector<int> v;
v.reserve(1000000); // but this might be more efficient
for (size_t i = 0; i != 1000000; ++i) { v.push_back(i); }
```

- Note the difference between `reserve` and `resize`.

Iterators May Become Invalidated, Other Notes

- When elements are inserted or removed before the end of the vector, iterators pointing to elements after the insertion points are invalidated.
- When an insertion causes a vector's capacity to grow, all iterators are invalidated.
- If you really need to shrink a vector's capacity to its current size, use the function `shrink_to_fit` C++11. (It's just a request that the system should get rid of excess capacity.)
- `vector<bool>` isn't a "normal" vector, since the `bool` values are bits that are packed (similar to the class `Bitset` in lab 4).

Deques

- The class `deque` (double-ended queue, pronounced "deck") is implemented as an array of pointers to arrays of elements.
- A deque is like a vector but also provides operations for insertions and removals at the front:

  `push_front(t)` Insert a copy of `t` at the front.
  `pop_front()` Remove the first element.

- Deques don't have the `capacity` and `reserve` operations of vectors. Instead, memory grows and shrinks as necessary. Any insertion or removal may cause reallocation and consequently invalidation of iterators.

Lists

- The class `list` is implemented as a doubly-linked list. Insertion and removal anywhere is fast.
- As `deque`, `list` supports `push_front` and `pop_front`.
- `list` iterators are bidirectional (not random access).
- In addition to the usual sequence operations, `list` has `splice` operations to move part of a list to another list and efficient `sort` and `merge` operations. You cannot use the library `sort` algorithm, since it needs random-access iterators.

Don't Use copy for Insertions

When we discussed iterators, we repeatedly used copy and back_inserter to insert elements in a vector. There are better ways to do this, which avoid the repeated calls of push_back.

```
array<int, 1000> x = { 1, 2, 3, ..., 1000 };

// copy, 1000 push_backs
vector<int> v;
copy(x.begin(), x.end(), back_inserter(v));

// insert first allocates memory, then copies
vector<int> v;
v.insert(v.end(), x.begin(), x.end());

// same as insert, but at initialization
vector<int> v(x.begin(), x.end());
```

Stacks

A `stack` is, like `queue` and `priority_queue`, implemented using one of the sequence classes. The `stack` interface (simplified):

```
template <typename T, typename Container = deque<T>>
class stack {
protected:
    Container c;
public:
    stack() {}
    explicit stack(const Container& s) : c(s) {}
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    T& top() { return c.back(); }
    const T& top() const { return c.back(); }
    void push(const T& t) { c.push_back(t); }
    void pop() { c.pop_back(); }
};
```

Why Both top() and pop()?

One might wonder why a stack has both `top()` and `pop()`, instead of the more usual `pop()` that returns and removes the top value. There are two reasons: 1) the latter variant isn't exception safe, 2) the latter variant is inefficient (it must return a value that must be copied):

```
value_type pop() {
    value_type ret = c.back();
    c.pop_back();
    return ret;
}
```

The same reasoning explains why the containers have both `back()` and `pop_back()` operations.

Associative Containers

An associative container contains elements that are identified by a key.

- A map stores key/value pairs, a set stores only the keys.
- A multimap and a multiset allow multiple elements with the same key.
- There are two versions of each class: one “ordered” (implemented using a search tree), one “unordered” (implemented using a hash table). Both versions have the same interface.
- The key type in an ordered container must support the < operator (used for sorting).
- There must be a hash function for the key type in an unordered container. The standard types have hash functions.
- The unordered containers are new in `C++11`.

Creating maps and sets

To create a map, you must specify the type of the key and of the corresponding value. To create a set, you must specify the key type.

```
map<string, int> days_in_month; // key = month name,
                                // value = days in month

set<string> used_emails;        // key = e-mail address

unordered_map<string, string> phonebook; // key = name,
                                // value = phone number
```

Elements in maps

The elements (key/value pairs) in maps are objects of the library class `pair`. The pair's components are stored in the public members `first` and `second`. Examples:

```
pair<int, double> p1; // both components value-initialized
pair<int, double> p2(1, 2.5);
...
cout << p2.first << "," << p2.second << endl;
```

The map `insert` operation takes a pair as parameter. It is convenient to create the pair on the fly:

```
map<string, int> daysInMonth;
days_in_month.insert(pair<string, int>("January", 31));
days_in_month.insert(make_pair("February", 28));
days_in_month.insert({"March", 31}); // new in C++11
```

Operations

In addition to the common container operations, associative containers have the following operations:

- **insert(t)** Insert a copy of `t` and return an iterator pointing to the new element. For a unique container, the return value is a pair with the iterator as the first component and a bool indicating success or failure as the second component.
- **`find(k)`** Return an iterator pointing to the element with key `k`, or `end()`.
- **`count(k)`** Return the number of elements with key `k`.
- **`equal_range(k)`** Return a range containing all elements with key `k`, as a pair `p` of iterators. The range is `[p.first,p.second)`.
- **`erase(k)`** Destroy and remove all elements with the key `k`, return the number of erased elements.
- **`erase(p)`** Destroy and remove the element pointed to by `p`.

Inserting and Finding in a map

Read words, count number of occurrences, print in sorted order:

```
int main() {
    ifstream in("input.txt");
    map<string, size_t> word_count;
    string word;
    while (in >> word) {
        auto it = word_count.find(word);
        if (it == word_count.end()) {
            word_count.insert(make_pair(word, 1));
        } else {
            it->second++;
        }
    }
    for (const auto& w : word_count) {
        cout << w.first << "\t" << w.second << endl;
    }
}
```

Using operator[] With Maps

In the map classes, [] is overloaded so the map functions as an associative array. For a map m, m[key] returns a reference to the associated value if the key exists in the map. Otherwise, it inserts the key and a default-constructed value and returns a reference to it.

The code from the previous slide to count the number of word occurrences can be written in the following way:

```
string word;
while (in >> word) {
    ++word_count[word];
}
```

Only use [] if you really want the insertion to take place if the key doesn't exist.

Use map-Specific Functions, Not Library Algorithms

`map::find` uses a tree search with logarithmic time complexity to find an element in a map, `unordered_map::find` uses a constant-time hash search. If you used the library `find` algorithm, the search time would be linear. Compare:

```
auto it = word_count.find("Hello");
if (it != words.end()) {
    cout << it->first << " occurs " << it->second << " times";
}

it = find_if(word_count.begin(), word_count.end(),
    [](const pair<string,int>& e) { return e.first == "Hello"; });
if (it != words.end()) { ... }
```

Using an Unordered Container

If the key type of an unordered container is non-standard, you must supply a hash function and an equality operator. Example:

```
class Person {... string get_name() const; ... };

struct Hash {
    size_t operator()(const Person& p) const {
        return std::hash<string>()(p.get_name());
    }
};

struct Eq {
    bool operator()(const Person& p1, const Person& p2) const {
        return p1.get_name() == p2.get_name();
    }
};

unordered_map<Person, string, Hash, Eq> phone_book;
```