Constructors

A constructor initializes a newly instantiated object

- a class can have multiple constructors
  - they differ in the arguments that they accept
  - which one is invoked depends on how the object is instantiated

You can write constructors for your object

- but if you don’t write any, C++ might automatically synthesize a default constructor for you
  - the default constructor is one that takes no arguments and that calls default constructors on all non-POD member variables
  - C++ does this iff your class has no const or reference data members, and no other user-defined constructors
Constructors, continued

You might choose to define multiple constructors:

```cpp
Point::Point() {
    x_ = 0;
    y_ = 0;
}

Point::Point(const int x, const int y) {
    x_ = x;
    y_ = y;
}

void foo() {
    Point x;   // invokes the default (argument-less) constructor
    Point y(1,2);   // invokes the two-int-arguments constructor
}
```
Constructors, continued

You might choose to define only one:

```cpp
Point::Point(const int x, const int y) {
    x_ = x;
    y_ = y;
}

void foo() {
    // Compiler error; if you define any constructors, C++ will
    // not automatically synthesize a default constructor for you.
    Point x;

    // Works.
    Point y(1,2); // invokes the two-int-arguments constructor
}
```
Initialization lists

As shorthand, C++ lets you declare an initialization list as part of your constructor declaration

- initializes fields according to parameters in the list
- the following two are (nearly) equivalent:

```cpp
Point::Point(const int x, const int y) : x_(x), y_(y) {
    std::cout << "Point constructed: (" << x_ << ",";
    std::cout << y_ << ")" << std::endl;
}
```

```cpp
Point::Point(const int x, const int y) {
    x_ = x;
    y_ = y;
    std::cout << "Point constructed: (" << x_ << ",";
    std::cout << y_ << ")" << std::endl;
}
```
Initialization vs. construction

```cpp
#ifndef _POINT_H_
#define _POINT_H_

class Point {
  public:
    Point(const int x, const int y, const int z) :
      x_(x), y_(y) {
        z_ = z;
    }

  private:
    int x_, y_, z_;  
};  // class Point

#endif  // _POINT_H_
```
Initialization vs. construction

```cpp
#ifndef _POINT_H_
#define _POINT_H_

class Point {
    public:
        Point(const int x, const int y, const int z) :
            x_(x), y_(y) {
            z_ = z;
        }

    private:
        int x_, y_, z_; // class Point

    #endif // _POINT_H_
```

First, initialization list is applied.
Initialization vs. construction

```cpp
#ifndef __POINT_H__
#define __POINT_H__

class Point {
public:
    Point(const int x, const int y, const int z) :
        x_(x), y_(y) {
        z_ = z;
    }

private:
    int x_, y_, z_;  
}; // class Point

#endif // __POINT_H__
```

next, constructor is executed
Initialization vs. construction

When a new object is created using some constructor:
- first, the initialization list is applied to members
  ▶ in the order that those members appear within the class definition, not the order in the initialization list (!)
- next, the constructor is invoked, and any statements within it are executed

Prefer initialization to assignment
- Objects are already initialized by some constructor before they can be assigned - initializer avoids two separate steps
Copy constructors

C++ has the notion of a **copy constructor**
- used to create a new object as a copy of an existing object

```cpp
Point::Point(const int x, const int y) : x_(x), y_(y) {}  
Point::Point(const Point &copyme) {  // copy constructor  
    x_ = copyme.x_;  
    y_ = copyme.y_;  
}

void foo() {  
    // invokes the two-int-arguments constructor  
    Point x(1,2);  

    // invokes the copy constructor to construct y as a copy of x  
    Point y(x);   // could also write as “Point y = x;”
}  
```
When do copies happen?

The copy constructor is invoked if:

- you pass an object as a parameter to a call-by-value function

```cpp
void foo(Point x) { ... }
Point y; // default cons.
foo(y); // copy cons.
```

- you return an object from a function

```cpp
Point foo() {
    Point y; // default cos.
    return y; // copy cons.
}
```

- you initialize an object from another object of the same type

```cpp
Point x; // default cons.
Point y(x); // copy cons.
Point z = y; // copy cons.
```
But...the compiler is smart...

It sometimes uses a “return by value optimization” to eliminate unnecessary copies

- sometimes you might not see a constructor get invoked when you expect it

```cpp
Point foo() {
    Point y; // default constructor.
    return y; // copy constructor? optimized?
}

Point x(1,2); // two-ints-argument constructor.
Point y = x; // copy constructor.
Point z = foo(); // copy constructor? optimized?
```
Synthesized copy constructor

If you don’t define your own copy constructor, C++ will synthesize one for you

- it will do a shallow copy of all of the fields (i.e., member variables) of your class

- sometimes the right thing, sometimes the wrong thing

see SimplePoint.cc, SimplePoint.h
assignment != construction

The "=" operator is the assignment operator
- assigns values to an existing, already constructed object
- you can overload the "=" operator

```cpp
Point w;   // default constructor.
Point x(1,2);  // two-ints-argument constructor.
Point y = w;   // copy constructor.
y = x;        // assignment operator.
```
Overloading the "=" operator

You can choose to overload the "=" operator
- but there are some rules you should follow

```cpp
Point &Point::operator=(const Point& rhs) { 
  if (this != &rhs) { // always check against this
    x_ = rhs.x_;  
    y_ = rhs.y_;  
  }
  return *this;  // always return *this from = 
}
```

```cpp
Point a;       // default constructor
a = b = c;    // works because "=" returns *this
a = (b = c);  // equiv to above, as "=" is right-associative
(a = b) = c;  // works because "=" returns a non-const
```
Synthesized assignment oper.

If you don’t overload the assignment operator, C++ will synthesize one for you

- it will do a shallow copy of all of the fields (i.e., member variables) of your class

- sometimes the right thing, sometimes the wrong thing

see SimplePoint.cc, SimplePoint.h
Destructors

C++ has the notion of a destructor

- invoked automatically when a class instance is deleted / goes out of scope, etc., even via exceptions or other causes
- place to put cleanup code - free any dynamic storage or other resources owned by the object
- standard C++ idiom for managing dynamic resources
  - Slogan: “Resource Acquisition Is Initialization” (RAII)

```cpp
Point::~Point() {  // destructor
    // do any cleanup needed when a Point object goes away
    // (nothing to do here since we have no dynamic resources)
}
```
Rule of Three

If you define any of:

1. Destructor
2. Copy Constructor
3. Assignment (operator=)

Then you should normally define all three
Dealing with the insanity

C++ style guide tip

- if possible, disable the copy const. and assignment operator

  ‣ *not possible if you want to store objects of your class in an STL container, unfortunately*

```cpp
class Point {
public:
  Point(int x, int y) : x_(x), y_(y) { }

private:
  // disable copy cons. and "=" by declaring but not defining
  Point(Point &copyme);
  Point &operator=(Point &rhs);
};

Point w;  // compiler error
Point x(1,2);  // OK
Point y = x;  // compiler error
x = w;  // compiler error
```
Dealing with the insanity

C++ style guide tip
- if you disable them, then you should instead have an explicit “CopyFrom” function

class Point {
public:
    Point::Point(int x, int y) : x_(x), y_(y) { }
    void CopyFrom(const Point &copy_from_me);

private:
    // disable copy cons. and "=" by declaring but not defining
    Point(const Point &copyme);
    Point &operator=(const Point &rhs);
};

Point x(1,2);  // OK
Point y(3,4);  // OK
x.CopyFrom(y);  // OK
new

To allocate on the heap using C++, you use the “new” keyword instead of the “malloc()” stdlib.h function

- you can use new to allocate an object
- you can use new to allocate a primitive type

To deallocate a heap-allocated object or primitive, use the “delete” keyword instead of the “free()” stdlib.h function

- if you’re using a legacy C code library or module in C++
  ‣ if C code returns you a malloc()’d pointer, use free() to deallocate it
  ‣ **never** free() something allocated with new
  ‣ **never** delete something allocated with malloc()
C++11 nullptr

C and C++ have long used NULL as a pointer value that references nothing

C++11 introduced a new literal for this: nullptr

- New reserved word
- Interchangeable with NULL for all practical purposes
  ‣ But it has type T* for any/every T, and is not an integer value
    • Avoids funny edge cases (see C++ references for details)
    • Still can convert to/from integer 0 for tests, assignment, etc.
- Advice: prefer nullptr in C++11 code (but NULL will also be around for a long, long time)
Dynamically allocated arrays

To dynamically allocate an array
- use "type *name = new type[size];" 

To dynamically deallocate an array
- use "delete[] name;"
- it is an error to use "delete name;" on an array
  ‣ the compiler probably won’t catch this, though!!!
  ‣ it can’t tell if it was allocated with “new type[size];” or “new type;”

see arrays.cc
## malloc vs. new

<table>
<thead>
<tr>
<th></th>
<th>malloc( )</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>what is it</strong></td>
<td>a function</td>
<td>an operator and keyword</td>
</tr>
<tr>
<td><strong>how often used in C</strong></td>
<td>often</td>
<td>never</td>
</tr>
<tr>
<td><strong>how often used in C++</strong></td>
<td>rarely</td>
<td>often</td>
</tr>
<tr>
<td><strong>allocates memory for</strong></td>
<td>anything</td>
<td>arrays, structs, objects, primitives</td>
</tr>
<tr>
<td><strong>returns</strong></td>
<td>a (void *) <em>(needs a cast)</em></td>
<td>appropriate pointer type <em>(doesn’t need a cast)</em></td>
</tr>
<tr>
<td><strong>when out of memory</strong></td>
<td>returns NULL</td>
<td>throws an exception</td>
</tr>
<tr>
<td><strong>deallocating</strong></td>
<td>free</td>
<td>delete or delete[ ]</td>
</tr>
</tbody>
</table>
Overloading the "==" operator

Remember the rules we should follow?

- here’s why; hugely subtle bug

```cpp
Foo::Foo(int val) { Init(val); }  
Foo::~Foo() { delete my_ptr_; }

void Foo::Init(int val) { my_ptr_ = new int; *my_ptr_ = val; }

Foo &Foo::operator=(const Foo& rhs) {
   // bug...we forgot our "if (self == &rhs) { ... }" guard
   delete my_ptr_;  
   Init(*((rhs.my_ptr_))); // might crash here (see below)
   return *this; // always return *this from =
}

void bar() {
   Foo a(10);  // default constructor 
   Foo b(20);  // default constructor 
   a = a;     // crash above; dereference delete’d pointer!!
}
```
Overloading the "=" operator

Remember the rules we should follow?

This is yet another reason for disabling the assignment operator, when possible
Suppose that...

You want to write a function to compare two ints:

```cpp
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int &value1, const int &value2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```
Suppose that...

You want to write a function to compare two ints, and you also want to write a function to compare two strings:

```cpp
// note the cool use of function overloading!

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int &value1, const int &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string &value1, const string &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```
Hmm....

The two implementations of compare are nearly identical.
- we could write a compare for every comparable type
  - but, that’s obviously a waste; lots of redundant code!

Instead, we’d like to write “generic code”
- code that is type-independent
- code that is compile-time polymorphic across types
C++: parametric polymorphism

C++ has the notion of templates

- a function or class that accepts a type as a parameter
  - you implement the function or class once, in a type-agnostic way
  - when you invoke the function or instantiate the class, you specify (one or more) types, or values, as arguments to it
- at compile-time, when C++ notices you using a template...
  - the compiler generates specialized code using the types you provided as parameters to the template
Function template

You want to write a function to compare two things:

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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <class T>
int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare<std::string>(h, w) << std::endl;
    std::cout << compare<int>(10, 20) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    return 0;
}
```

Function template

Same thing, but letting the compiler infer the types:

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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <class T>
int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare(10, 20) << std::endl;
    std::cout << compare("Hello", "World") << std::endl;  // bug!
    std::cout << compare(h, w) << std::endl;  // ok
    return 0;
}
```

`functiontemplate_infer.cc`
You can use non-types (constant values) in a template:

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

template <class T, int N>
void printmultiple(const T &value1) {
    for (int i = 0; i < N; ++i)
        std::cout << value1 << std::endl;
}

int main(int argc, char **argv) {
    std::string h("hello");
    printmultiple<std::string, 3>(h);
    printmultiple<const char *, 4>("hi");
    printmultiple<int, 5>(10);
    return 0;
}
```
What’s going on underneath?

The compiler doesn’t generate any code when it sees the templated function

- it doesn’t know what code to generate yet, since it doesn’t know what types are involved

When the compiler sees the function being used, then it understands what types are involved

- it generates the instantiation of the template and compiles it
  - the compiler generates template instantiations for each type used as a template parameter
  - kind of like macro expansion
This creates a problem...

```cpp
#ifndef _COMPARE_H_
#define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b);

#endif // COMPARE_H__

#include "compare.h"

template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}

#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

compare.cc

```cpp
#include "compare.h"

template <class T>
int comp(const T& a, const T& b);

#endif // COMPARE_H__
```

```cpp
#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

main.cc

```cpp
#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

main.cc
One solution

```cpp
#ifndef _COMPARE_H_
define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}

#endif // COMPARE_H_
```

```cpp
#include <iostream>
#include "compare.h"
using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

Another solution

```cpp
#ifndef _COMPARE_H_
define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b);
#include "compare.cc"
#endif // COMPARE_H_

#include <iostream>
#include "compare.h"
using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

```cpp
# ifndef _COMPARE_H_
# define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```

```cpp
#include <iostream>
#include "compare.h"
using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```
Class templates

Templating is useful for classes as well! Imagine we want a class that holds a pair of things

- we want to be able to:
  
  ▶ set the value of the first thing, second thing
  ▶ get the value of the first thing, second thing
  ▶ reverse the order of the things
  ▶ print the pair of things
#include <iostream>
#include <string>

template <class Thing> class Pair {
public:
    Pair() { };
    
    Thing &get_first() { return first_; }
    Thing &get_second();
    void set_first(Thing &copyme);
    void set_second(Thing &copyme);
    void Reverse();

private:
    Thing first_, second_; 
};

#include "Pair.cc"
Pair class

template <class Thing> Thing &Pair<Thing>::get_second() {  
  return second_;  
}

template <class Thing> void Pair<Thing>::set_first(Thing &copyme) {  
  first_ = copyme;  
}

template <class Thing> void Pair<Thing>::set_second(Thing &copyme) {  
  second_ = copyme;  
}

template <class Thing> void Pair<Thing>::Reverse() {  
  // makes *3* copies  
  Thing tmp = first_;  
  first_ = second_;  
  second_ = tmp;  
}
Pair class

```cpp
#include <iostream>
#include <string>
#include "Pair.h"

int main(int argc, char **argv) {
    Pair<std::string> ps;
    std::string x("foo"), y("bar");
    ps.set_first(x);
    ps.set_second(y);
    ps.Reverse();
    std::cout << ps.get_first() << std::endl;
    return 0;
}
```
C++’s standard library

Consists of four major pieces:

- the entire C standard library
- C++’s input/output stream library
  ‣ std::cin, std::cout, stringstreams, fstreams, etc.
- C++’s standard template library (STL)
  ‣ containers, iterators, algorithms (sort, find, etc.), numerics
- C++’s miscellaneous library
  ‣ strings, exceptions, memory allocation, localization

http://www.cplusplus.com/reference/
Containers!

- a container is an object that stores (in memory) a collection of other objects (elements)
  - implemented as class templates, so hugely flexible
- several different classes of container
  - sequence containers (vector, deque, list)
  - associative containers (set, map, multiset, multimap, bitset)
- differ in algorithmic cost, supported operations
STL :(

STL containers store by value, not by reference

- when you insert an object, the container makes a copy

- if the container needs to rearrange objects, it makes copies
  - e.g., if you sort a vector, it will make many many copies
  - e.g., if you insert into a map, that may trigger several copies

- what if you don’t want this (disabled copy con, or copy $$)?
  - you can insert a wrapper object with a pointer to the object
  - we’ll learn about these “smart pointers” later
STL vector

A generic, dynamically resizable array

- elements are stored in contiguous memory locations
  - elements can be accessed using pointer arithmetic if you like
  - random access is $O(1)$ time
- adding / removing from the end is cheap (constant time)
- inserting / deleting from middle or start is expensive ($O(n)$)
STL iterator

Each container class has an associated iterator class

- used to iterate through elements of the container (duh!)
- some container iterators support more operations than others
  - all can be incremented (++ operator), copied, copy-cons’ed
  - some can be dereferenced on RHS (e.g., x = *it;)
  - some can be dereferenced on LHS (e.g., *it = x;)
  - some can be decremented (-- operator)
  - some support random access ([ ], +, -, +=, -=, <, > operators)

http://www.cplusplus.com/reference/std/iterator/
STL algorithms

A set of functions to be used on ranges of elements

- range: any sequence that can be accessed through iterators or pointers, like arrays or some of the containers

- algorithms operate directly on values using assignment or copy constructors, rather than modifying container structure

- some do not modify elements
  - find, count, for_each, min_element, binary_search, etc.

- some do modify elements
  - sort, transform, copy, swap, etc.

http://www.cplusplus.com/reference/algorithm/
STL list

A generic doubly-linked list

- elements are *not* stored in contiguous memory locations
  - does not support random access (cannot do list[5])
- some operations are much more efficient than vectors
  - constant time insertion, deletion anywhere in list
  - can iterate forward or backwards
- has a built-in sort member function
  - no copies; manipulates list structure instead of element values

http://www.cplusplus.com/reference/stl/list/
STL map

A key/value table, implemented as a tree

- elements stored in sorted order
  - key value must support less-than operator
- keys must be unique
  - multimap allows duplicate keys
- efficient lookup ($O(\log n)$) and insertion ($O(\log n)$)

http://www.cplusplus.com/reference/stl/map/
New in C++ 11

unordered_map, unordered_set
- and related classes: unordered_multimaps, unordered_multisets
- average case for key access is $O(1)$
  - But range iterators can be less efficient than ordered map/set
- See C++ Primer, online references for details