# Large-Scale C++: Advanced *Levelization* Techniques

John Lakos

Tuesday, September 20, 2016

*This version is for CppCon'16, Bellevue, WA.*

# Copyright Notice

© 2014 Bloomberg L.P. Permission is granted to copy, distribute, and display this material, and to make derivative works and commercial use of it. The information in this material is provided "AS IS", without warranty of any kind. Neither Bloomberg nor any employee guarantees the correctness or completeness of such information. Bloomberg, its employees, and its affiliated entities and persons shall not be liable, directly or indirectly, in any way, for any inaccuracies, errors or omissions in such information. Nothing herein should be interpreted as stating the opinions, policies, recommendations, or positions of Bloomberg.

# Abstract

**Developing a large-scale software system in C++ requires more than just a sound understanding of the logical design issues covered in most books on C++ programming. To be successful, one also needs a grasp of physical design concepts that, while closely tied to the technical aspects of development, include a dimension with which even expert software developers may have little or no experience.**

**In this talk we begin by briefly reviewing the basics of physical design. We then present a variety of *levelization* and *insulation* techniques, and apply them in present-day, real-word examples to avoid cyclic, excessive, or otherwise inappropriate physical dependencies. Along the way, we comment on how to make the best use of what the C++ language has to offer.**

# What’s The Problem?

# What’s The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

# What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

- It involves many subtle *logical* and *physical* aspects.

# What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

- It involves many subtle *logical* and *physical* aspects.
- It requires an ability to isolate and modularize functionality within discrete, fine-grained physical components.

# What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

- It involves many subtle *logical* and *physical* aspects.
- It requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.
- It requires the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.

# What's The Problem?

Large-Scale C++ Software Design is Multi-Dimensional:

- It involves many subtle *logical* and *physical* aspects.
- It requires an ability to isolate and modularize logical functionality within discrete, fine-grained physical components.
- It requires the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.
- It requires attention to numerous logical and physical rules that govern sound software design.

# Purpose of this Talk

# Purpose of this Talk

1. Review the basics of component-based design:

# Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component
   - Our Three-Level Physical-Packaging Hierarchy
   - Logical *Encapsulation* Versus Physical *Insulation*
   - When to `#include` a Header File in a Header

# Purpose of this Talk

1. Review the basics of component-based design:
   - Component Properties and Logical Diagrams
   - Implied Dependency and Level Numbers
   - The Two Most Important Physical Design Rules
   - Guidelines for Collocating Classes in a Component
   - Our Three-Level Physical-Packaging Hierarchy
   - Logical *Encapsulation* Versus Physical *Insulation*
   - When to `#include` a Header File in a Header
2. Introduce a variety of *levelization* and *insulation* techniques, and then apply them to present-day real-world examples to avoid cyclic, excessive, or otherwise inappropriate physical dependencies.

# Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Survey of Advanced *Levelization* Techniques

   Avoiding Cyclic, Excessive, or Inappropriate Dependencies

3. Total and Partial *Insulation* Techniques

   Avoiding Unnecessary Compile-Time Coupling

4. Present-Day, Real-World Design Examples

   Prioritizing Physical Design to Improve Software Quality

# Outline

1. Review of Elementary Physical Design
   - Components, Modularity, Physical Dependencies
2. Survey of Advanced *Levelization* Techniques
   - Avoiding Cyclic, Excessive, or Inappropriate Dependencies
3. Total and Partial *Insulation* Techniques
   - Avoiding Unnecessary Compile-Time Coupling
4. Present-Day, Real-World Design Examples
   - Prioritizing Physical Design to Improve Software Quality

# Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

# Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

Logical: Classes and Functions

# Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

Logical: Classes and Functions

Physical: Files and Libraries

# Logical versus Physical Design

*Logical* content aggregated into a *Physical* hierarchy of ***components***

# *Component:* Uniform Physical Structure

## A Component Is Physical

```
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
  // ...


}
//-- END OF FILE --
```

**component.t.cpp**

```
// component.h

// ...




//-- END OF FILE --
```

**component.h**

```
// component.cpp
#include <component.h>
// ...




//-- END OF FILE --
```

**component.cpp**

**component**

# *Component:* Uniform Physical Structure

## Implementation

```
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
  // ...


}
//-- END OF FILE --
```

**component.t.cpp**

```
// component.h

// ...



//-- END OF FILE --
```

**component.h**

```
// component.cpp
#include <component.h>
// ...



//-- END OF FILE --
```

**component.cpp**

**component**

# *Component:* Uniform Physical Structure

# *Component:* Uniform Physical Structure

## Test Driver

```
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
  // ...


}
//-- END OF FILE --
```

**component.t.cpp**

```
// component.h

// ...



//-- END OF FILE --
```

**component.h**

```
// component.cpp
#include <component.h>
// ...



//-- END OF FILE --
```

**component.cpp**

**component**

# *Component:* Uniform Physical Structure

## The Fundamental Unit of Design

```
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
  // ...


}
//-- END OF FILE --
```

**component.t.cpp**

```
// component.h

// ...



//-- END OF FILE --
```

**component.h**

```
// component.cpp
#include <component.h>
// ...



//-- END OF FILE --
```

**component.cpp**

**component**

# *Component:* Not Just a `.h`/`.cpp` Pair

**`my_widget`**

# *Component:* Not Just a `.h`/`.cpp` Pair

**There are four Properties…**

# *Component:* Not Just a `.h`/`.cpp` Pair

1. The **`.cpp`** file includes its **`.h`** file as the first substantive line of code.

**EVEN IF THE .CPP IS OTHERWISE EMPTY!**

# *Component:* Not Just a `.h`/`.cpp` Pair

1. The `.cpp` file includes its `.h` file as the first substantive line of code.
2. All logical constructs having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.

# *Component:* Not Just a `.h`/`.cpp` Pair

1. The `.cpp` file includes its `.h` file as the first substantive line of code.
2. All logical constructs having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.
3. All constructs having external or dual *bindage* declared in a `.h` file (if defined at all) are defined within the component.

# *Component:* Not Just a `.h`/`.cpp` Pair

1. The `.cpp` file includes its `.h` file as the first substantive line of code.
2. All logical constructs having external *linkage* defined in a `.cpp` file are declared in the corresponding `.h` file.
3. All constructs having external or dual *bindage* declared in a `.h` file (if defined at all) are defined within the component.
4. A component's functionality is accessed via a **`#include`** of its header, and never via a forward (**`extern`**) declaration.

# Logical Relationships

# Logical Relationships

Is-A

# Logical Relationships

# Logical Relationships

Uses-in-the-Interface

Is-A

# Logical Relationships

# Logical Relationships

Uses-in-the-Interface

Is-A

# Logical Relationships

Uses-in-the-Interface

Uses-in-the-Implementation

Is-A

# Logical Relationships

Uses-in-the-Interface

Uses-in-the-Implementation

Is-A

# Logical Relationships

Uses-in-the-Interface

Uses-in-the-Implementation

Is-A

# Logical Relationships

Uses-in-the-Interface

Uses-in-the-Implementation

Uses in name only

Is-A

# Logical Relationships

Uses-in-the-Interface

Uses-in-the-Implementation

Uses in name only

Is-A

# Implied Dependency

Uses-in-the-Interface

Uses-in-the-Implementation

Uses in name only

Is-A

# Implied Dependency

Uses-in-the-Interface

Uses-in-the-Implementation

Depends-On

Uses in name only

Is-A

# Implied Dependency

# Implied Dependency

# Implied Dependency

# Implied Dependency

# Level Numbers

# Level Numbers

# Level Numbers

# Level Numbers

# Level Numbers

# Essential Physical Design Rules

# Essential Physical Design Rules

There are two:

# Essential Physical Design Rules

There are two:

1. <u>No</u> *Cyclic* Physical Dependencies!

# Essential Physical Design Rules

There are two:

1. No *Cyclic* Physical Dependencies!

2. No *Long-Distance* Friendships!

# Criteria for Colocating “Public” Classes

# Criteria for Colocating “Public” Classes

There are four:

# Criteria for Colocating “Public” Classes

There are four:

1. Friendship.

# Criteria for Colocating “Public” Classes

There are four:

1. Friendship.

2. Cyclic Dependency.

# Criteria for Colocating "Public" Classes

There are four:

1. Friendship.
2. Cyclic Dependency.
3. Single Solution.

# Criteria for Colocating "Public" Classes

Single Solution

# Criteria for Colocating "Public" Classes

Single Solution

Hierarchy of Solutions

# Criteria for Colocating "Public" Classes

Single Solution

Hierarchy of Solutions

# Criteria for Colocating "Public" Classes

Single Solution

Hierarchy of Solutions

# Criteria for Colocating "Public" Classes

Single Solution

Hierarchy of Solutions

# Criteria for Colocating "Public" Classes

Single Solution

Hierarchy of Solutions

# Criteria for Colocating "Public" Classes

Single Solution

Hierarchy of Solutions

# Criteria for Colocating "Public" Classes

# Criteria for Colocating "Public" Classes

Single Solution

Hierarchy of Solutions

# Criteria for Colocating "Public" Classes

There are four:

1. Friendship.
2. Cyclic Dependency.
3. Single Solution.
4. "Flea on an Elephant."

# Criteria for Colocating "Public" Classes

"Flea on an Elephant"

(Elephant on a Flea)

# Criteria for Colocating "Public" Classes

"Flea on an Elephant"

(Elephant on a Flea)

# Criteria for Colocating "Public" Classes

"Flea on an Elephant"

(Elephant on a Flea)

# Criteria for Colocating "Public" Classes

"Flea on an Elephant" (Elephant on a Flea)

# Physical Dependency

Five levels of **physical dependency**:

Level 5:

Level 4:

Level 3:

Level 2:

Level 1:

# Physical Aggregation

Only one level of **physical aggregation**:

# The Package

## Two levels of physical aggregation:

**"A Hierarchy of Component Hierarchies"**

# The Package

## Two levels of physical aggregation:

**Metadata governs, even absent of any components!**

# The Package

## Two levels of physical aggregation:

**Metadata governs allowed dependencies.**

# Package Dependencies

## Aggregate dependencies:

# The Package Group

# Non-Uniform Physical-Aggregation Depth

1 1 & 2 1, 2, & 3 1, 2, 3, & 4 deep 1, 2, 3, 4, & 5 deep

# Non-Uniform Physical-Aggregation Depth

Less Regular;
Harder To Understand;
Less Conducive To Reuse

1 1 & 2 1, 2, & 3 1, 2, 3, & 4 deep 1, 2, 3, 4, & 5 deep

# Non-Uniform Physical-Aggregation Depth

1 1 & 2 1, 2 & 3 1, 2, & 4 deep 1, 2, 3, 4, & 5 deep

# Uniform Depth of Physical Aggregation

Component Package Package Group

# Uniform Depth of Physical Aggregation

**Exactly Three Levels of Physical Aggregation**

Component Package Package Group

# Uniform Depth of Physical Aggregation

# Uniform Depth of Physical Aggregation

# Uniform Depth of Physical Aggregation

Component Package Package Group Package Group Hierarchy

# End of Section

# Questions?

# What Questions are we Answering?

- What distinguishes *Logical* from *Physical* Design?
- What are the fundamental properties of a `.h/.cpp` pair that make it a component?
- How do we infer physical relationships (*Depends-On*) from logical relationships (e.g., *Is-A* and *Uses*)?
- What are *level numbers*, and how do we determine them?
- How do we extract component dependencies efficiently?
- What essential physical design rules must be followed?
- What are the criteria for colocating classes & functions?
- How are components aggregated into larger physical units?
- How many levels of physical aggregation do we employ?

# What Questions are we Answering?

- What distinguishes *Logical* from *Physical* Design?
- What are the fundamental properties of a `.h/.cpp` pair that make it a component?
- How do we infer physical relationships (*Depends-On*) from logical relationships (e.g., *Is-A* and *Uses*)?
- What are *level numbers*, and how do we determine them?
- How do we extract component dependencies efficiently?
- What essential physical design rules must be followed?
- What are the criteria for colocating classes & functions?
- How are components aggregated into larger physical units?
- How many levels of physical aggregation do we employ?

# Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Survey of Advanced *Levelization* Techniques

   Avoiding Cyclic, Excessive, or Inappropriate Dependencies

3. Total and Partial *Insulation* Techniques

   Avoiding Unnecessary Compile-Time Coupling

4. Present-Day, Real-World Design Examples

   Prioritizing Physical Design to Improve Software Quality

# Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Survey of Advanced *Levelization* Techniques

   Avoiding Cyclic, Excessive, or Inappropriate Dependencies

3. Total and Partial *Insulation* Techniques

   Avoiding Unnecessary Compile-Time Coupling

4. Present-Day, Real-World Design Examples

   Prioritizing Physical Design to Improve Software Quality

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

Usage:

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

Usage:

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

## Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

## Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.
- Are you sure that design is *levelizable* – i.e., do we know how to make its physical dependencies acyclic?

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

## Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.
- Are you sure that design is *levelizable* – i.e., do we know how to make its physical dependencies acyclic?

# Levelization

Levelize (v.); Levelizable (a.); Levelization (n.)

## Usage:

- We need to *levelize* that design – i.e., we need to make its physical dependency graph acyclic.
- Are you sure that design is *levelizable* – i.e., do we know how to make its physical dependencies acyclic?
- What *levelization* techniques would you use – i.e., what techniques would you use to *levelize* your design?

# Levelization

Levelize (v.); Levelizable (a.); Level

Usage:

- We need to
  its
- A i.e., do we know
  h dependencies acyclic?
- W techniques would you use – i.e., what
  tec would you use to *levelize* your design?

(Shameless)

Advertisement

Note that Lakos'96 described 9 different ways to untangle cyclic physical dependencies: ***Escalation, Demotion, Opaque Pointers, Dumb Data, Redundancy, Callbacks, Manager Class, Factoring, and Escalating Encapsulation.***

# Levelization

Levelize (v.); Levelizable (a.); Level

Usage

- cyclic?
- techniques would you use – i.e., what techniques would you use to *levelize* your design?

**Let's learn these techniques now!**

Note that Lakos'96 described 9 different ways to untangle cyclic physical dependencies: ***Escalation, Demotion, Opaque Pointers, Dumb Data, Redundancy, Callbacks, Manager Class, Factoring, and Escalating Encapsulation.***

# Levelization

Levelize (v.); Levelizable (a.); Level

**Later, we'll apply them to real-world examples.**

- techniques would you use – i.e., what techniques would you use to *levelize* your design?

Note that Lakos'96 described 9 different ways to untangle cyclic physical dependencies: ***Escalation, Demotion, Opaque Pointers, Dumb Data, Redundancy, Callbacks, Manager Class, Factoring, and Escalating Encapsulation.***

# Escalation

***Escalation*** – Moving mutually dependent functionality higher in the physical hierarchy.

# Escalation

# Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  Point d_origin;
  int   d_width;
  int   d_length;
 public:
  // …
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  Point d_lowerLeft;
  Point d_upperRight;

 public:
  // …
  Box(const Rect& r);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  Point d_origin;
  int   d_width;
  int   d_length;
 public:
  // …
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  Point d_lowerLeft;
  Point d_upperRight;

 public:
  // …
  Box(const Rect& r);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& ll,
       int w, int h);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

Implicit Conversions

# Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
#include <box.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
class Rect;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
};
```

```
// box.h
#include <point.h>
class Rect;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(const Rect& r);
  // …
};
```

# Escalation

# Escalation

# Escalation

# Escalation

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(Rect& r);
  // …
  operator Rect() const;
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
#include <rect.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  Box(Rect& r);
  // …
  operator Rect() const;
  // …
};
```

# Escalation

# Escalation

If rectangle.h is modified, *all* clients of Box will need to recompile.
Box
3
Rect
2
Point
1

# Escalation

# Escalation

```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
  operator Box() const;
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// rect.h
#include <point.h>
class Box;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  Rect(const Box& b);
  // …
  operator Box() const;
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

Rect
3
Box
2
Point
1
If box.h is modified,
*only* direct clients of Box
will need to recompile.

# Escalation

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to support *inline functions*

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to support *inline functions*

more on this later

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to avoid *transitive includes*

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(
  static Rect rectFromBox(
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect
  static Rect rectFromBox(
  // …
};

inline
ConvertUtil::boxFromRectangle(
                              const Rect& r)
{
  int length = r.ur.x() - r.ll.x();
  int width  = r.ur.y() - r.ll.y();
  Point origin(r.ll.x() + length/2,
               r.ll.y() + width/2);
  return Box(origin, length, width);
}
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(
  static Rect rectFromBox(
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

struct ConvertUtil {
  static Box boxFromRect(
  static Rect rectFromBox(
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
#include <point.h>
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
#include <point.h>
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

```
// rect.h
class Point;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
class Point;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>

class ConvertUtil {
  BoxFromRect(const Rect& r);
  RectFromBox(const Box& b);
```

```
//
class
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
  Box();
  Box(const Point& ul,
      const Point& ur);
  // …
};
```

**convertutil.h No Longer Compiles!**

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(
  static Rect rectFromBox(
  // …
};
```

in order to avoid *transitive includes*

Presumes `inline` implementations of *(elided)* functions using `Point`.

```
// rect.h
class Point;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
class Point;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to avoid *transitive includes*

```
// rect.h
class Point;
class Rect {
  //
 pub
  Re
  Re

  //
};
```

```
// box.h
class Point;
class Box {
```

**convertutil.h Again Compiles!**

# Escalation

```
// convertutil.h
#include <rect.h>
#include <box.h>
#include <point.h>
struct ConvertUtil {
  static Box boxFromRect(const Rect& r);
  static Rect rectFromBox(const Box& b);
  // …
};
```

in order to avoid *transitive includes*

```
// rect.h
class Point;
class Rect {
  // …
 public:
  Rect();
  Rect(const Point& o,
       int w, int l);
  // …
};
```

```
// box.h
class Point;
class Box {
  // …
 public:
  Box();
  Box(const Point& ll,
      const Point& ur);
  // …
};
```

# Escalation

# Escalation

convertutil.h
convertutil.cpp
convertutil
rect.h
rect.cpp
rect
box.h
box.cpp
box
point.h
point.cpp
point

## Escalation

# Discussion?

# Demotion

***Demotion*** – Moving common functionality lower in the physical hierarchy.

# Demotion

# Demotion

# Demotion

```
// eventqueue.h
#include <event.h>
#include <list.h>
class EventQueue {
  List<Event> d_events;

  // common event info.

  // other stuff

 public:
  // …

};
```

```
// event.h
class Event {
  EventQueue *d_common_p;

  // event-specific
  // information

 public:
  // …
  int commonData() const;
  // …
};
```

# Demotion

```
// eventqueue.h
#include <event.h>
#include <list.h>
class EventQueue {
  List<Event> d_events;

  // common event info.

  // other stuff

 public:
  // …

};
```

```
// event.h
class Event {
  EventQueue *d_common_p;

  // event-specific
  // information

 public:
  // …
  int commonData() const;
  // …
};
```

# Demotion

```
// eventqueue.h
#include <event.h>
#include <list.h>
class EventQueue {
  List<Event> d_events;

  // common event info.

  // other stuff

 public:
  // …

};
```

```
// event.h
class Event {
  EventQueue *d_common_p;

  // event-specific
  // information

 public:
  // …
  int commonData() const;
  // …
};
```

# Demotion

common
event info.
other stuff
EventQueue
event-
specific
info.
Event
event-
specific
info.
Event
event-
specific
info.
Event

# Demotion

# Demotion

# Demotion

# Demotion

# Demotion

# Demotion

```
// eventqueue.h
#include <event.h>
#include <list.h>
class EventQueue {
  List<Event> d_events;

  // common event info.

  // other stuff

 public:
  // …

};
```

```
// event.h
class Event {
  EventQueue *d_common_p;

  // event-specific
  // information

 public:
  // …
  int commonData() const;
  // …
};
```

# Demotion

```
// eventqueue.h
#include <event.h>
#include <list.h>
#include <commoneventinfo.h>
class EventQueue {
  List<Event> d_events;
  CommonEventInfo d_info;
  // other stuff
 public:
  // …
};
```

```
// event.h
#include <commoneventInfo.h>
class Event {
  CommonEventInfo *d_common_p;
  // event-specific
  // information
 public:
  // …
  int commonData() const;
  // …
};
```

```
// commoneventinfo.h
class CommonEventInfo {
  // common event info.
 public:
  // …
};
```

# Demotion

common
event info.
other stuff
EventQueue
event-
specific
info.
Event
event-
specific
info.
Event
event-
specific
info.
Event

# Demotion

d_info
other stuff
EventQueue
event-specific info.
Event
event-specific info.
Event
event-specific info.
Event

# Demotion

## Demotion

# Discussion?

# Opaque Pointers

***Opaque Pointers*** **–** Having an object use another *in name only*.

# Opaque Pointers

# Opaque Pointers

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
#include <manager.h>
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  int numStaff() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
#include <manager.h>
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  int numStaff() const;
};
```

```
#include <employee.h>
int askEmployeeNumStaff(const Employee& employee) {
  return employee.numStaff();
}
```

# Opaque Pointers

# Opaque Pointers

# Opaque Pointers

# Opaque Pointers

# Opaque Pointers

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
#include <manager.h>
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  int numStaff() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
#include <manager.h>
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  int numStaff() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  int numStaff() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  int numStaff() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  //
  int numStaff() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  //
  const Manager
              *manager() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  // …
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
             *manager() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
      employee(int id) const;
  // …
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
              *manager() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  int numStaff() const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
              *manager() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  int numStaff() const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
             *manager() const;
};
```

# Opaque Pointers

# Opaque Pointers

# Opaque Pointers

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  int numStaff() const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
             *manager() const;
};
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  int numStaff() const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
             *manager() const;
};
```

```
// client.cpp
#include <employee.h>
#include <manager.h>
int askEmployeeNumStaff(const Employee& employee) {
  return employee.manager()->numStaff();
}
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  int numStaff() const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
             *manager() const;
};
```

```
// client.cpp
#include <employee.h>
#include <manager.h>
int askEmployeeNumStaff(const Employee& employee) {
  return employee.manager()->numStaff();
}
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  int numStaff() const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
            *manager() const;
};
```

```
// client.cpp
#include <employee.h>
#include <manager.h>
int askEmployeeNumStaff(const Employee& employee) {
  return employee.manager()->numStaff();
}
```

# Opaque Pointers

```
// manager.h
#include <employee.h>
#include <vector>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
  int numStaff() const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  const Manager
            *manager() const;
};
```

```
// client.cpp
#include <employee.h>
#include <manager.h>
int askEmployeeNumStaff(const Employee& employee) {
  return employee.manager()->numStaff();
}
```

## Opaque Pointers

# Discussion?

# Dumb Data

***Dumb Data*** – Using data that indicates a dependency on a peer object, but only in the context of a separate, higher-level object.

# Dumb Data

# Dumb Data

# Dumb Data

```
// node.h
#include <vector>
#include <edge.h>
class Node {
  vector<Edge *> d_edges;
  // …
 public:
  Node();
  // …
  void addEdge(Edge *edge);
  // …
  int numEdges() const;
  // …
  const Edge *edge(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h
#include <node.h>
class Edge {
  Node *d_head_p;
  Node *d_tail_p;
  // …
 public:
  Edge(Node *head, Node *tail);
  // …
  void setWeight(double value);
  // …
  const Node *head() const;
  const Node *tail() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

```
// node.h
#include <vector>
#include <edge.h>;
class Node {
  vector<Edge *> d_edges;
  // …
 public:
  Node();
  // …
  void addEdge(Edge *edge);
  // …
  int numEdges() const;
  // …
  const Edge *edge(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h
#include <node.h>
class Edge {
  Node *d_head_p;
  Node *d_tail_p;
  // …
 public:
  Edge(Node *head, Node *tail);
  // …
  void setWeight(double value);
  // …
  const Node *head() const;
  const Node *tail() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

```
// node.h
#include <vector>
class Edge;
class Node {
  vector<Edge *> d_edges;
  // …
 public:
  Node();
  // …
  void addEdge(Edge *edge);
  // …
  int numEdges() const;
  // …
  const Edge *edge(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h
class Node;
class Edge {
  Node *d_head_p;
  Node *d_tail_p;
  // …
 public:
  Edge(Node *head, Node *tail);
  // …
  void setWeight(double value);
  // …
  const Node *head() const;
  const Node *tail() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

```
// node.h
#include <vector>
class Edge;
class Node {
  vector<Edge *> d_edges;
  // …
 public:
  Node();
  // …
  void addEdge(Edge *edge);
  // …
  int numEdges() const;
  // …
  const Edge *edge(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h
class Node;
class Edge {
  Node *d_head_p;
  Node *d_tail_p;
  // …
 public:
  Edge(Node *head, Node *tail);
  // …
  void setWeight(double value);
  // …
  const Node *head() const;
  const Node *tail() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

# Dumb Data

# Dumb Data

```
// node.h
#include <vector>
class Edge;
class Node {
  vector<Edge *> d_edges;
  // …
 public:
  Node();
  // …
  void addEdge(Edge *edge);
  // …
  int numEdges() const;
  // …
  const Edge *edge(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h
class Node;
class Edge {
  Node *d_head_p;
  Node *d_tail_p;
  // …
 public:
  Edge(Node *head, Node *tail);
  // …
  void setWeight(double value);
  // …
  const Node *head() const;
  const Node *tail() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

```
// node.h
#include <vector>
class Edge;
class Node {
  vector<Edge *> d_edges;
  // …
 public:
  Node();
  // …
  void addEdge(Edge *edge);
  // …
  int numEdges() const;
  // …
  const Edge *edge(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h
class Node;
class Edge {
  Node *d_head_p;
  Node *d_tail_p;
  // …
 public:
  Edge(Node *head, Node *tail);
  // …
  void setWeight(double value);
  // …
  const Node *head() const;
  const Node *tail() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

```
// node.h
#include <vector>

class Node {
  vector<int> d_edgeIds;
  // …
 public:
  Node();
  // …
  void addEdge(int edgeId);
  // …
  int numEdges() const;
  // …
  int edgeId(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h

class Edge {
  int d_headId;
  int d_tailId;
  // …
 public:
  Edge(int headId, int tailId);
  // …
  void setWeight(double value);
  // …
  int headId() const;
  int tailId() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

```
// node.h
#include <vector>

class Node {
  vector<int> d_edgeIds;
  // …
 public:
  Node();
  // …
  void addEdge(int edgeId);
  // …
  int numEdges() const;
  // …
  int edgeId(int idx) const;
  // …
  int value() const;
  // …
};
```

```
// edge.h

class Edge {
  int d_headId;
  int d_tailId;
  // …
 public:
  Edge(int headId, int tailId);
  // …
  void setWeight(double value);
  // …
  int headId() const;
  int tailId() const;
  // …
  double weight() const;
  // …
};
```

# Dumb Data

```
// graph.h
#include <node.h>
#include <edge.h>
#include <vector>
class Graph {
  vector<Node> d_nodes;
  vector<Edge> d_edges;
 public:
  Graph();
  // …
  const Node *addNode();
  const Edge *addEdge(const Node *tail, const Node *head);
  // …
  int numNodes() const;
  int numEdges() const;
  const Node *node(int nodeId);
  const Edge *edge(int edgeId);
};
```

# Dumb Data

```
// graph.h
#include <node.h>
#include <edge.h>
#include <vector>
class Graph {
  vector<Node> d_nodes;
  vector<Edge> d_edges;
 public:
  Graph();
  // …
  const Node *addNode();
  const Edge *addEdge(const Node *tail, const Node *head);
  // …
  int numNodes() const;
  int numEdges() const;
  const Node *node(int nodeId);
  const Edge *edge(int edgeId);
};
```

# Dumb Data

```
// graph.h
#include <node.h>
#include <edge.h>
#include <vector>
class Graph {
  vector<Node> d_nodes;
  vector<Edge> d_edges;
 public:
  Graph();
  // …
  const Node *addNode();
  const Edge *addEdge(const Node *tail, const Node *head);
  // …
  int numNodes() const;
  int numEdges() const;
  const Node *node(int nodeId);
  const Edge *edge(int edgeId);
};
```

# Dumb Data

## Opaque Pointers

```
// client.h
#include <graph.h>
#include <node.h>
#include <edge.h>

double firstEdgeWeight(const Node *node)
{
  return node->edge(0)->weight();
}

int headValue(const Edge *edge)
{
  return edge->head()->value();
}

int firstAdjacentValue(const Node *node)
{
  return node->edge(0)->head()->value();
}
```

# Dumb Data

**Opaque Pointers**

```
// client.h
#include <graph.h>
#include <node.h>
#include <edge.h>

double firstEdgeWeight(const Node *node)
{
  return node->edge(0)->weight();
}

int headValue(const Edge *edge)
{
  return edge->head()->value();
}

int firstAdjacentValue(const Node *node)
{
  return node->edge(0)->head()->value();
}
```

# Dumb Data

## Opaque Pointers

```
double firstEdgeWeight(const Node *node)
{
  return node->edge(0)->weight();
}
```

# Dumb Data

## Opaque Pointers

```
double firstEdgeWeight(const Node *node)
{
  return node->edge(0)->weight();
}
```

## Dumb Data

```
double firstEdgeWeight(const Graph& g, const Node *node)
{
  return g.edge(node->edgeId(0))->weight();
}
```

# Dumb Data

Dumb Data

```
// client.h
#include <graph.h>
#include <node.h>
#include <edge.h>

double firstEdgeWeight(const Graph& g, const Node *node)
{
  return g.edge(node->edgeId(0))->weight();
}

int headValue(const Graph& g, const Edge *edge)
{
  return g.node(edge->headId())->value();
}

int firstAdjacentValue(const Graph& g, const Node *node)
{
  return g.node(g.edge(node->edgeId(0))->headId())->value();
}
```

# Dumb Data

```
// client.h
#include <graph.h>
#include <node.h>
#include <edge.h>

double firstEdgeWeight(const Graph& g, const Node *node)
{
  return g.edge(node->edgeId(0))->weight();
}

int headValue(const Graph& g, const Edge *edge)
{
  return g.node(edge->headId())->value();
}

int firstAdjacentValue(const Graph& g, const Node *node)
{
  return g.node(g.edge(node->edgeId(0))->headId())->value();
}
```

**Dumb Data**

**Node and Edge are Independently Externalizable!**

## Dumb Data

# Discussion?

# Redundancy

***Redundancy*** **–** Deliberately avoiding reuse by repeating a small amount of code or data to avoid coupling.

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

```
bool isLeapYear(int year)
{
  return 0 == year % 4 && 0 != year % 100 || 0 == year % 400;
}
```

# Redundancy

```
bool isProlepticGregorianLeapYear(int year)
{
  return 0 == year % 4 && 0 != year % 100 || 0 == year % 400;
}
```

## 2. Survey of Advanced *Levelization* Techniques

# Redundancy

```
inline int min(int a, int b)
{
  return a < b ? a : b;
}
```

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

**Demotion is Mandatory for Vocabulary Types!**

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

Well-Packaged,
Complete,
Generally Useful
Functionality
2
1

# Redundancy

Well-Packaged,
Complete,
Generally Useful
Functionality
2
1
?

# Redundancy

Well-Packaged, Complete, Generally Useful Functionality
2
Requires Only Tiny Subset of Higher-Level Functionality
1
?

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

# Redundancy

## Redundancy

# Discussion?

# Callbacks

***Callbacks*** – Client-supplied functions/data that enable lower-level subsystems to perform specific tasks in a more global context.

# Callbacks

There are several flavors:

1. DATA (Effectively *Demotion*)
2. FUNCTION (Stateless/Stateful)
3. FUNCTOR (Function Object)
4. PROTOCOL (Abstract Interface)
5. CONCEPT (Structural Interface)

# Callbacks: DATA

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;
 public:
  // …
  int addEmployee(…);
  const Employee&
     employee(int id) const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss, …);
  // …
  int numStaff() const;
};
```

# Callbacks: DATA

# Callbacks: DATA

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;

 public:
  // …
  int addEmployee();
  const Employee&
     employee(int id) const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss);
  // …
  int numStaff() const;
};
```

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;

 public:
  // …
  int addEmployee();
  const Employee&
     employee(int id) const;
};
```

```
// employee.h
class Manager;
class Employee {
  Manager *d_boss_p;
 public:
  Employee(Manager *boss);
  // …
  int numStaff() const;
};
```

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;

 public:
  // …
  int addEmployee();
  const Employee&
     employee(int id) const;
};
```

```
// employee.h

class Employee {
  int *const d_numStaff_p;
 public:
  Employee(int *nStaffAddr);
  // …
  int numStaff() const;
};
```

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;

 public:
  // …
  int addEmployee();
  const Employee&
     employee(int id) const;
};
```

```
// employee.h

class Employee {
  int *const d_numStaff_p;
 public:
  Employee(int *nStaffAddr);
  // …
  int numStaff() const;
};

inline int
Employee::numStaff() const {
  return *d_numStaff_p;
}
```

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;
  int d_nStaff; // init to 0
 public:
  // …
  int addEmployee();
  const Employee&
     employee(int id) const;
};
```

```
// employee.h

class Employee {
  int *const d_numStaff_p;
 public:
  Employee(int *nStaffAddr);
  // …
  int numStaff() const;
};

inline int
Employee::numStaff() const {
  return *d_numStaff_p;
}
```

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;
  int d_nStaff; // init to 0
 public:
  // …
  int addEmployee();
  const Employee&
     employee(int id) const;
};
inline int
Manager::addEmployee() {
  d_staff.push_back(&d_nStaff);
  return d_nStaff++;
}
```

```
// employee.h

class Employee {
  int *const d_numStaff_p;
 public:
  Employee(int *nStaffAddr);
  // …
  int numStaff() const;
};

inline int
Employee::numStaff() const {
  return *d_numStaff_p;
}
```

# Callbacks: DATA

```
// manager.h
#include <employee.h>
#include <vector.h>
class Manager {
  vector<Employee> d_staff;
  int d_nStaff; // init to 0
 public:
  // …
  int addEmployee();
  const Employee&
     employee(int id) const;
};
inline int
Manager::addEmployee() {
  d_staff.push_back(&d_nStaff);
  return d_nStaff++;
}
```

```
// employee.h

class Employee {
  int *const d_numStaff_p;
 public:
  Employee(int *nStaffAddr);
  // …
  int numStaff() const;
};

inline int
Employee::numStaff() const {
  return *d_numStaff_p;
}
```

**Effectively Demotion!**

# Callbacks: DATA

**Effectively *Demotion!***

# Callbacks: DATA

**Effectively *Demotion!***

# Callbacks: DATA

"Variation in Value"

—Tom Cargill (c. 1992)

**Effectively *Demotion!***

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  // Do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  // Do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  // Do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  // Do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// …
// …
static MySta  globalState;
void saveAn  ()
{
   // Fi  save w k.
   //
   s  .:exit(1);
}

int main()
{
   // Do useful wor
   // …

   return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  // Do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
static void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

```
// main.cpp
// …
static MyState globalState;
static void saveAndExit()
{
  // First save work.
  // …
  std::exit(1);
}

int main()
{
  set_new_handler(saveAndExit);

  // Now do useful work.
  // …

  return 0;
}
```

# Callbacks: Function

# Callbacks: Function

```
// eventmgr.h
#incude <event.h>
class EventMgr {
  // …
 public:
  // …
  void schedule(Event *event);
};

inline void EventMgr::invoke()
{
  d_head->invoke();
  // …
}
```

```
// event.h
#include <eventmgr.h>
class Event {
  // …
 public:
  // …
  void schedule(EventMgr *eMgr);
  void invoke();
};
inline
void schedule(EventMgr *eMgr)
{
  eMgr->schedule(this);
}
```

# Callbacks: Function

```
// eventmgr.h
#incude <event.h>
class EventMgr {
  // …
 public:
  // …
  void schedule(Event *event);
};

inline void EventMgr::invoke()
{
  d_head->invoke();
  // …
}
```

```
// event.h
#include <eventmgr.h>
class Event {
  // …
 public:
  // …
  void schedule(EventMgr *eMgr);
  void invoke();
};
inline
void schedule(EventMgr *eMgr)
{
  eMgr->schedule(this);
}
```

# Callbacks: Function

```
// eventmgr.h
#incude <event.h>
class EventMgr {
  // …
 public:
  // …
  void schedule(Event *event);
};

inline void EventMgr::invoke()
{
  d_head->invoke();
  // …
}
```

```
// event.h
#include <eventmgr.h>
class Event {
  // …
 public:
  // …
  void schedule(EventMgr *eMgr);
  void invoke();
};
inline
void schedule(EventMgr *eMgr)
{
  eMgr->schedule(this);
}
```

# Callbacks: Function

```
// anydarnthing.h
#include <eventmgr.h>
class AnyDarnThing {
  static void invoke(void *me);
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Function

```
// anydarnthing.h
#include <eventmgr.h>
class AnyDarnThing {
  static void invoke(void *me);
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
// …

void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Function

```
// anydarnthing.h
#include <eventmgr.h>
class AnyDarnThing {
  static void invoke(void *me);
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
// …

void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Function

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  static void invoke(void *me);
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Function

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  static void invoke(void *me);
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  static void invoke(void *me);
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  static void invoke(void *me);
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  void invoke();
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  void invoke();
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef void (*EventCb)(void *);
  // …
  void schedule(EventCb  cb
                void    *data);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  void invoke();
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef
    std::function<void()>& EventCb;
  // …
  void schedule(EventCb cb);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  void invoke();
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(invoke, this);
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef
    std::function<void()>& EventCb;
  // …
  void schedule(EventCb cb);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  void invoke();
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(bind(
     &AnyDarnThing::invoke, this));
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef
    std::function<void()>& EventCb;
  // …
  void schedule(EventCb cb);
};
```

# Callbacks: Functor

```
// anydarnthing.h
class EventMgr;
class AnyDarnThing {
  void invoke();
 public:
  // …
  void schedule(EventMgr *eMgr);
}
```

```
// anydarnthing.cpp
#include <eventmgr.h>
// …
void AnyDarnThing::schedule(
                EventMgr *eMgr) {
  eMgr->schedule(
          [this]{ invoke(); } );
}
```

```
// eventmgr.h
// …
class EventMgr {
  // …
 public:
  typedef
    std::function<void()>& EventCb;
  // …
  void schedule(EventCb cb);
};
```

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

```
// gameof21.h
#include <player.h>
class GameOf21 {
// …
 public:
  // …
  addPlayer(const Player *p,
            int          cash);
  void dealCards();
  void shuffleCards();
  // …
  const Deck& deck() const;
  const Rules& rules() const;
};
```

```
// player.h
#include <gameof21.h>
class Player {
  // …
 public:
  Player(const char *name);
  // …
  int bet(const GameOf21&
                  game) const;
  Action choice(const GameOf21&
                  game) const;
  const char *name() const;
};
```

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

```
// player.h
class GameOf21;
class Player {
  // …
 public:
  virtual ~Player();
  virtual int bet(
    const GameOf21& g) const = 0;
  virtual Action choice(
    const GameOf21& g) const = 0;
  virtual const char *name()
                          const = 0;
};
```

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Protocol

# Callbacks: Concept

# Callbacks: Concept

Externalization

# Callbacks: Concept

Externalization

# Callbacks: Concept

# Callbacks: Concept

# Callbacks: Concept

```
// bytestream.h
#include <charbuf.h>
class ByteStream {
  CharBuf d_streamData;
  // …
 public:
  // …
  void putInt8(int value);
  void putInt16(int value);
  // …
  void putFloat32(double value);
  void putFloat64(double value);
  // …
};
```

```
// charbuf.h
#include <bytestream.h>
class CharBuf {
  char *d_buffer_p;
  // …
 public:
  // …
  void appendChar(char c);
  size_t length() const;
  char operator[](size_t i) const;
  // …
  void
  streamOut(ByteStream *s) const;
};
```

# Callbacks: Concept

```
// bytestream.h
#include <charbuf.h>
class ByteStream {
  CharBuf d_streamData;
  // …
 public:
  // …
  void putInt8(int value);
  void putInt16(int value);
  // …
  void putFloat32(double value);
  void putFloat64(double value);
  // …
};
```

```
// charbuf.h
#include <bytestream.h>
class CharBuf {
  char *d_buffer_p;
  // …
 public:
  // …
  void appendChar(char c);
  size_t length() const;
  char operator[](size_t i) const;
  // …
  void
  streamOut(ByteStream *s) const;
};
```

# Callbacks: Concept

```
// bytestream.
#include <cha uf.h>
class ByteStr am {
  CharBuf d_ reamData;
  // …
 public:
  // …
  void putInt (int value);
  void putInt (int value
  // …
  void putFloat (dou e value);
  void putFloat64 ble value);
  // …
};
```

```
// cha uf.h
#inc ue <bytestr m.h>
cl s CharBuf {
  char *d_buffer_p
  // …
 public:
  // …
  void appendChar ar c);
  size_t length() onst;
  char operator[] ize_t i) const;
  // …
  void
  streamOut teStream *s) const;
};
```

# Callbacks: Concept

# Callbacks: Concept

# Callbacks: Concept

```
// stream.h
class Stream {
 public:
  virtual ~Stream();
  // …
  virtual void
        putInt8(int value) = 0;
  virtual void
        putInt16(int value) = 0;
  // …
};
```

# Callbacks: Concept

```
// stream.h
class Stream {
 public:
  virtual ~Stream();
  // …
```

**Too #@%& Slow!!**

# Callbacks: Concept

```
// stream.h
class Stream {
 public:
  virtual ~Stream();
  // …
};
```

**Too #@%& Slow!!**

# Callbacks: Concept

# Callbacks: Concept

# Callbacks: Concept

# Callbacks: Concept

# Callbacks: Concept

# Callbacks: Concept

# Callbacks: Concept

```
// bytestream.h
#include <charbuf.h>
class ByteStream {
  CharBuf d_streamData;
  // …
 public:
  // …
  void putInt8(int value);
  void putInt16(int value);
  // …
  void putFloat32(double value);
  void putFloat64(double value);
  // …
};
```

```
// charbuf.h
class CharBuf {
  // …
 public:
  // …
  void appendChar(char c);
  // …
  template<class STREAM> void
  streamOut(STREAM *s) const;
};
```

# Callbacks: Concept

```
// bytestream.h
#include <charbuf.h>
class ByteStream {
  CharBuf d_streamData;
  // …
 public:
  // …
  void putInt8(int value);
  void putInt16(int value);
  // …
  void putFloat32(double value);
  void putFloat64(double value);
  // …
};
```

```
// charbuf.h
class CharBuf {
  // …
 public:
  // …
  void appendChar(char c);
  // …
  template<class STREAM> void
  streamOut(STREAM *s) const;
};
```

**CharBuf is Externalizable!**

## Callbacks

# Discussion?

# Manager Class

***Manager Class*** **–** *Establishing a class that owns and coordinates lower-level objects.*

# Manager Class

## List

"List??"

# Manager Class

# Manager Class

## List

# Manager Class

## List

# Manager Class

```
// link.cpp
// …
Link::~Link()
{
  delete d_next_p;
}
```

# Manager Class

```
// link.cpp
// …
Link::~Link()
{
  delete d_next;
}
```

# Manager Class

## List

# Manager Class

## List

# Manager Class

## List

# Manager Class

## List

# Manager Class

## List

# Manager Class

## List

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

# Manager Class

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

## Graph

# Manager Class

# Discussion?

# Factoring

***Factoring*** – Moving independently testable sub-behavior out of the implementation of a complex component involved in excessive physical coupling.

# Factoring

# Factoring

# Factoring

# Factoring

# Factoring

Architecturally Significant Client-Facing Interface!

# Factoring

Architecturally Significant Client-Facing Interface!

Private methods above become public ones in this class as sole data member.

# Factoring

Architecturally Significant Client-Facing Interface!

Private methods above become public ones in this class as sole data member.

File-scope `static` methods in `.cpp` file are made public in separate utility `struct`.

# Factoring

# Factoring

# Factoring

# Factoring

```
// date.h
#include <dateimputil.h>
class Date {
  int d_serialDate;
 public:
  Date(int y, int m, int d);
  // …
  void setYmd(int y, int m, int d};
  int setYmdIfValid(int y, int m, int d);
  // …
  bool isLeapyear() const;
};
```

# Factoring

```
// date.h
#include <dateimputil.h>
class Date {
  int d_serialDate;
 public:
  Date(int y, int m, int d);
  // …
  void setYmd(int y, int m, int d};
  int setYmdIfValid(int y, int m, int d);
  // …
  bool isLeapyear() const;
};
```

```
// dateimputil.h
// …
struct DateImpUtil {
  static int toSerialDate(int y, int m, int d);
  // …
  static bool isLeapYear(int y);
};
```

# Factoring

# Factoring

# Factoring

# Factoring

# Factoring

# Factoring

# Factoring

1
Network
Machine
Cable

1
MachineImp

1
NetworkImp

1
CableImp

# Factoring

1
Network
Machine
Cable
MachineImp
1
NetworkImp
1
CableImp
1

# Factoring

2
Network
Machine
Cable
MachineImp
1
NetworkImp
1
CableImp
1

# Factoring

2
Network
Machine
Cable
1
MachineImp
1
NetworkImp
1
CableImp

# Factoring

Network
2
Machine
Cable
MachineImp
1
NetworkImp
1
CableImp
1

## Factoring

# Discussion?

## Escalating Encapsulation

***Escalating Encapsulation***

– Moving the point at which implementation details are hidden from clients to a higher level in the physical hierarchy.

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Multi-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Single-Component Wrapper

# Escalating Encapsulation

## Hiding Header Files

# Escalating Encapsulation

## Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

# Escalating Encapsulation

## Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

```
// right.h
#include <cookie.h>
class Right {
  // …
  const Cookie& getC() const;
  // …
};
```

# Escalating Encapsulation

## Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

```
// right.h
#include <cookie.h>
class Right {
  // …
  const Cookie& getC() const;
  // …
};
```

# Escalating Encapsulation

## Hiding Header Files

```
// left.h
#include <cookie.h>
class Left {
  // …
  void setC(const Cookie& c);
  // …
};
```

```
// right.h
#include <cookie.h>
class Right {
  // …
  const Cookie& getC() const;
  // …
};
```

# Escalating Encapsulation

## Hiding Header Files

Bad Idea:

Hide `cookie.h`?

# Escalating Encapsulation

## Hiding Header Files

Bad Idea:

(1) Convolves architecture with deployment.

# Escalating Encapsulation

## Hiding Header Files

Bad Idea:

(1) Convolves architecture with deployment.

(2) Inhibits side-by-side reuse of the "hidden" component.

Hide `cookie.h`?

# Escalating Encapsulation

## Hiding Header Files

Bad Idea:

(1) Convolves architecture with deployment.

(2) Inhibits side-by-side reuse of the "hidden" component.

Hide `cookie.h`?

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

syst

# Escalating Encapsulation

## Wrapper Package

**wrap**

**syst**

# Escalating Encapsulation

## Wrapper Package

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_left.h
// …
class wrap_Left {
  syst_Left d_imp;
 public: // …
```

```
// wrap_right.h
// …
class wrap_Right {
  syst_Right d_imp;
 public: // …
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```

# Escalating Encapsulation

## Wrapper Package

```
// wrap_right.cpp
// …
void wrap_Right::someFunction(const wrap_Left& v) {
  const syst_Left& vImp = *reinterpret_cast<syst_Left *>(&v);
  d_imp.someFunction(vImp);
```

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

Package naming is more than just a convention:

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

# Escalating Encapsulation

**sub** ⬅ Exactly Three Characters

# Escalating Encapsulation

# Discussion?

# End of Section

# Questions?

# Levelization Techniques (Summary)

***Escalation*** – Moving mutually dependent functionality higher in the physical hierarchy.

***Demotion*** – Moving common functionality lower in the physical hierarchy.

***Opaque Pointers*** – Having an object use another *in name only*.

***Dumb Data*** – Using data that indicates a dependency on a peer object, but only in the context of a separate, higher-level object.

***Redundancy*** – Deliberately avoiding reuse by repeating a small amount of code or data to avoid coupling.

***Callbacks*** – Client-supplied functions/data that enable lower-level subsystems to perform specific tasks in a more global context.

***Manager Class*** – Establishing a class that owns and coordinates lower-level objects.

***Factoring*** – Moving independently testable sub-behavior out of the implementation of a complex component involved in excessive physical coupling.

***Escalating Encapsulation*** – Moving the point at which implementation details are hidden from clients to a higher level in the physical hierarchy.

# Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Survey of Advanced *Levelization* Techniques

   Avoiding Cyclic, Excessive, or Inappropriate Dependencies

3. Total and Partial *Insulation* Techniques

   Avoiding Unnecessary Compile-Time Coupling

4. Present-Day, Real-World Design Examples

   Prioritizing Physical Design to Improve Software Quality

# Outline

1. Review of Elementary Physical Design
   - Components, Modularity, Physical Dependencies
2. Survey of Advanced *Levelization* Techniques
   - Avoiding Cyclic, Excessive, or Inappropriate Dependencies
3. Total and Partial *Insulation* Techniques
   - Avoiding Unnecessary Compile-Time Coupling
4. Present-Day, Real-World Design Examples
   - Prioritizing Physical Design to Improve Software Quality

# Insulation

# Insulation

*Logical* ***encapsulation*** versus *physical* ***insulation***:

# Insulation

*Logical* ***encapsulation*** versus *physical* ***insulation***:

- An implementation detail of a component (type, data, or function) that can be altered, added, or removed *without* forcing clients to <u>*rework their code*</u> is said to be ***encapsulated***.

# Insulation

*Logical* ***encapsulation*** versus *physical* ***insulation***:

- An implementation detail of a component (type, data, or function) that can be altered, added, or removed *without* forcing clients to *rework their code* is said to be ***encapsulated***.
- An implementation detail of a component (type, data, or function) that can be altered, added, or removed *without* forcing clients to *recompile* is said to be ***insulated***.

# Insulation

d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a

# Insulation

d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a

# Insulation

d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail
'D'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail
'D' is
Encapsulated
by 'C'
e.h
e.cpp
e
#include d
#include e
c.h
c.cpp
c
b.cpp
b
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail
(Use of) 'D' is
Encapsulated
by 'C'
e.h
e.cpp
e
#include d
#include e
c.h
c.cpp
c
b.cpp
b
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail 'E'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail 'E'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail 'E'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulation

Imp
Detail 'E'
d.h
d.cpp
d
e.h
e.cpp
e
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library
Component

# Insulat

'E' is Insulated by 'C'
d.h
d.cpp
d
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
Client
#include b
#include c
a.h
a.cpp
a
Library Component

# Insu

(Use of) 'E' is Insulated by 'C'
d.h
d.cpp
d
b.h
b.cpp
b
#include d
#include e
c.h
c.cpp
c
#include b
#include c
a.h
a.cpp
a
Client
Library Component

# Criteria for having a `#include` in a `.h` File

# Criteria for having a `#include` in a `.h` File

Recall that:

A header file must be "self-sufficient" w.r.t. compilation.

# Criteria for having a `#include` in a `.h` File

There are five:

# Criteria for having a `#include` in a `.h` File

There are five:

*1. Is-A*

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*

2. *Has-A*

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

*But not Uses !*

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

*But not Uses !*

```
#include <point.h>
Point appendVertex(int                index,
                   const Point& vertex);
```

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

*But not Uses !*

```
#include <point.h>
Point appendVertex(int                index,
                   const Point& vertex);
```

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

*But not Uses !*

```
class Point;
Point appendVertex(int                index,
                   const Point& vertex);
```

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

*But not Uses !*

```
#include <point.h>
Point appendVertex(int                index,
                   const Point& vertex);
```

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

*But not Uses !*

```
class Point;
Point appendVertex(int                index,
                   const Point& vertex);
```

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*
3. `inline` (used in function body)

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*
3. `inline` (used in function body)
4. `enum`

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*
3. `inline` (used in function body)
4. `enum`
5. `typedef` (e.g., template specialization)

# Criteria for having a `#include` in a `.h` File

There are five:

1. *Is-A*
2. *Has-A*
3. `inline` (used in function body)
4. `enum`
5. `typedef` (e.g., template specialization)

Note: Covariant return types is another edge case.

# Total Insulation Techniques

I. Pure Abstract Interface (Protocol Class)

II. Fully Insulating Concrete Class (“Pimple”)

III. Procedural Interface

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Pure Abstract Interface (Protocol Class)

# Discussion?

# Fully Insulating Concrete Class (“Pimple”)

What do we mean by “Pimple”?

# Fully Insulating Concrete Class ("Pimple")

What do we mean by "Pimple"?

"Pointer to

implementation"

# Fully Insulating Concrete Class (“Pimple”)

a.k.a. “Pimpl”

“Pointer to

implementation”

# Fully Insulating Concrete Class ("Pimple")

**concreteclass**

# Fully Insulating Concrete Class ("Pimple")

**concreteclass**

# Fully Insulating Concrete Class ("Pimple")

**concreteclass**

# Fully Insulating Concrete Class ("Pimple")

**concreteclass**

# Fully Insulating Concrete Class ("Pimple")

**concreteclass**

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

} Member Data

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

Creators

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

Creators

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

Creators

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

} Manipulators & Accessors

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

Accessors should be **`const`**-qualified.

Manipulators & Accessors

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …

class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

Free Operators

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {

  // …

};
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& ori
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
}
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {

  // …

};
```

Duplicate **All** Member Data and Creators!

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& ori
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
}
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

Duplicate **All** Member Data and Creators!

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  int d_data;  // some data
  // …
 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data;
// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

Replace Data with **SINGLE** Pointer to Imp.

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data;
// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

Replace Data with **SINGLE** Pointer to Imp.

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data)
  : d_data(data) { }
  MyClass(const MyClass& orig)
  : d_data(orig.d_data) { }
  ~MyClass() { /* XYZZY */ }
  // …
  int data() { return d_data; }
};
bool
operator==(const MyClass& a,
           const MyClass& b) {
  return a.d_data == b.d_data; }
// …
```

```
// m
#in
                                  ca)
  : d_

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

Eliminate **All** inline functions!

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data();
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// m
#in
                                  ca)
  : d_

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

Eliminate **All** inline functions!

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data();
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data();
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};
```

Now There's Room!

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

MyClass::MyClass(int data)
: d_imp_p(new MyClass_Imp(data))
{
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

MyClass::MyClass(int data)
: d_imp_p(new MyClass_Imp(data))
{
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass
//
st                                  ed
c

                                    ,
                                    b);

// …
```

(In Practice, We Would Now Also Pass an Optional Allocator at construction.)

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }

  Cla    Class(int data)
: d_imp_p(new MyClass_Imp(data))
{
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data,
       bslma::Allocator *a = 0);
  MyClass(const MyClass& orig,
       bslma::Allocator *a = 0);
  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>
#include <bslma_allocator.h>
struct MyClass_Imp {
  int d_data;
  // …
  bslma::Allocator *d_allocator_p;

  MyClass_Imp(int data,
              bslma::Allocator *a)
  : d_data(data)
  , d_allocator_p(a) { }
  // …
};

MyClass::MyClass(int data,
                 bslma::Allocator *a)
: d_imp_p(new(*a) MyClass_Imp(data, a))
{
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass
//
st                                  ed
c

                                    ,
                                    b);

// …
```

(We Will Ignore Allocators for Now.)

```
// myclass.cpp
#include <myclass.h>
#include <bslma_allocator.h>
struct MyClass_Imp {
  int d_data;
  // …
  bslma::Allocator *d_allocator_p;

  MyClass_Imp(int data,
              bslma::Allocator *a)
  : d_data(data)
  , d_allocator_p(a) { }
  // …
}

  as    yClass(int data,
              bslma::Allocator *a)
: d_imp_p(new(*a) MyClass_Imp(data, a))
{
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

MyClass::MyClass(int data)
: d_imp_p(new MyClass_Imp(data))
{
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

MyClass::MyClass(const MyClass& orig)
: d_imp_p(
     new MyClass_Imp(*orig.d_imp_p))
{
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

MyClass::~MyClass()
{
  delete d_imp_p;
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

int MyClass::data() const
{
  return d_imp_p->d_data;
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

int MyClass::data() const
{
  return d_imp_p->d_data;
}
```

# Fully Insulating Concrete Class ("Pimple")

Just access the data directly.

```
// my
//
s

 pub
  explicit My  (int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

int MyClass::data() const
{
  return d_imp_p->d_data;
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// myclass.cpp
#include <myclass.h>

struct MyClass_Imp {
  int d_data;
  // …
  explicit MyClass_Imp(int data)
  : d_data(data) { }

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

bool
MyClass::operator==(MyClass& a
                    MYClass& b)
{
  return a.d_imp_p->d_data
      == b.d_imp_p->d_data;
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
// …
struct MyClass_Imp; // insulated
class MyClass {
  MyClass_Imp *d_imp_p;

 public:
  explicit MyClass(int data);

  MyClass(const MyClass& orig);

  ~MyClass();
  // …
  int data() const;
};
bool
operator==(const MyClass& a,
           const MyClass& b);

// …
```

```
// my
#inc

  : d_da

  MyClass_Imp(const MyClass_Imp& orig)
  : d_data(orig.d_data) { }

  ~MyClass_Imp() { /* XYZZY */ }
};

bool
MyClass::operator==(MyClass& a
                    MYClass& b)
{
  return a.d_imp_p->d_data
      == b.d_imp_p->d_data;
}
```

One Extra Indirection!

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
```

```
// my
#inc
```

One Extra

**Not Architecturally Significant!**

```
// ...
```

```
}
```

# Fully Insulating Concrete Class ("Pimple")

```
// myclass.h
```

```
// my
#inc
```

One Extra

Not

Under No circumstances
Expose the Pointer
to the Implementation!

ant!

# Fully Insulating Concrete Class (“Pimple”)

# Discussion?

# Procedural Interface

# Procedural Interface

# Procedural Interface

# Procedural Interface

# Procedural Interface

Every procedural interface (PI) function is:

# Procedural Interface

Every procedural interface (PI) function is:

1. Entirely independent of every other PI function.

# Procedural Interface

Every procedural interface (PI) function is:

1. Entirely independent of every other PI function.
2. Physically separate from any C++ components.

# Procedural Interface

Every procedural interface (PI) function is:

1. Entirely independent of every other PI function.
2. Physically separate from any C++ components.
3. Totally devoid of additional implementation (beyond that needed for insulation purposes).

# Procedural Interface

Every procedural interface (PI) function is:

1. Entirely independent of every other PI function.
2. Physically separate from any C++ components.
3. Totally devoid of additional implementation (beyond that needed for insulation purposes).
4. Named in a natural, regular, and predictable way.

# Procedural Interface

Every procedural interface (PI) function is:

1. Entirely independent of every other PI function.
2. Physically separate from any C++ components.
3. Totally devoid of additional implementation (beyond that needed for insulation purposes).
4. Named in a natural, regular, and predictable way.
5. Callable from both C and C++.

# Procedural Interface

Every procedural interface (PI) function is:

1. Entirely independent of every other PI function.
2. Physically separate from any C++ components.
3. Totally devoid of additional implementation (beyond that needed for insulation purposes).
4. Named in a natural, regular, and predictable way.
5. Callable from both C and C++.
6. Exposing the actual underlying opaque C++ types.

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

```
// bdeg_point.h (continued)
inline
Point::Point()
: d_x(0), d_y(0) { }

inline
Point::Point(int x, int y)
: d_x(x), d_y(y) { }

inline
Point::Point(const Point& other)
: d_x(other.d_x)
, d_y(other.d_y) { }

inline
Point::~Point() { }

inline
Point& Point::operator=(const Point&
rhs) {
  d_x = rhs.d_x; d_y = rhs.d_y;
  return *this;
}
// …
```

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

Note that Internal Include Guards and Enterprise-Wide Namespace are Elided.

```
// bdeg_point.h (continued)
inline
Point::Point()
: d_x(0), d_y(0) { }

inline
Point::Point(int x, int y)
: d_x(x), d_y(y) { }

inline
Point::Point(const Point& other)
: d_x(other.d_x)
, d_y(other.d_y) { }

inline
Point::~Point() { }

inline
Point& Point::operator=(const Point&
rhs) {
  d_x = rhs.d_x; d_y = rhs.d_y;
  return *this;
}
// …
```

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

```
// bdeg_point.cpp
#include <bdeg_point.h>
#include <ostream>

namespace MyLongCompanyName {

namespace bdeg { // empty
} // package namespace

std::ostream&
bdeg::operator<<(std::ostream&
stream, const Point& point)
{
    return stream << '(' <<
        point.x() << ',' <<
        point.y() << ')' <<
        std::flush;
}

// …

} // enterprise namespace
```

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

```
// bdeg_point.cpp  <- Always Have .cpp
#include <bdeg_point.h>
#include <ostream>

namespace MyLongCompanyName {

namespace bdeg { // empty
} // package namespace

std::ostream&
bdeg::operator<<(std::ostream&
stream, const Point& point)
{
    return stream << '(' <<
        point.x() << ',' <<
        point.y() << ')' <<
        std::flush;
}

// …

} // enterprise namespace
```

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

```
// bdeg_point.cpp
#include <bdeg_point.h> <- Always First
#include <ostream>

namespace MyLongCompanyName {

namespace bdeg { // empty
} // package namespace

std::ostream&
bdeg::operator<<(std::ostream&
stream, const Point& point)
{
    return stream << '(' <<
        point.x() << ',' <<
        point.y() << ')' <<
        std::flush;
}

// …

} // enterprise namespace
```

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

```
// bdeg_point.cpp
#include <bdeg_point.h>
#include <ostream>
All our code resides in the Enterprise Namespace.
namespace MyLongCompanyName {

namespace bdeg { // empty
} // package namespace

std::ostream&
bdeg::operator<<(std::ostream&
stream, const Point& point)
{
    return stream << '(' <<
        point.x() << ',' <<
        point.y() << ')' <<
        std::flush;
}

// …

} // enterprise namespace
```

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

```
// bdeg_point.cpp
#include <bdeg_point.h>
#include <ostream>

namespace MyLongCompanyName {
Only members defined in package namespace!
namespace bdeg { // empty
} // package namespace

std::ostream&
bdeg::operator<<(std::ostream&
stream, const Point& point)
{
    return stream << '(' <<
        point.x() << ',' <<
        point.y() << ')' <<
        std::flush;
}

// …

} // enterprise namespace
```

# Procedural Interface

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

```
// bdeg_point.cpp
#include <bdeg_point.h>
#include <ostream>

namespace MyLongCompanyName {

namespace bdeg { // empty
} // package namespace
```

**Free operators/functions are defined outside.**

```
std::ostream&
bdeg::operator<<(std::ostream&
stream, const Point& point)
{
    return stream << '(' <<
        point.x() << ',' <<
        point.y() << ')' <<
        std::flush;
}

// …

} // enterprise namespace
```

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

The Z_ Prefix Provides One-To-One Mapping Between Procedural Wrappers and Corresponding C++ Components.

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                              int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                     z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                            int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h
#ifndef INCLUDED_Z_BDEG_POINT
#define INCLUDED_Z_BDEG_POINT
#ifdef __cplusplus
extern "C" {
  namespace MyLongCompanyName {
    namespace bdeg {
      class Point;
    } // package namespace
  } // enterprise namespace
  typedef
  MyLongCompanyName::bdeg::Point
  z_bdeg_Point;
#else
  typedef struct z_bdeg_Point
                 z_bdeg_Point;
#endif // internal include guard
```

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                            int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                     z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                              int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                              int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                 const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                      z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                       z_bdeg_Point *object,
                 const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
               z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
               z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
               const z_bdeg_Point *object);
int z_bdeg_Point_fY(
               const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                    z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                     z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                    z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                     z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                              int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                     z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                              int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                              int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                              int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                 const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                      z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                       z_bdeg_Point *object,
                 const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
               z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
               z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
               const z_bdeg_Point *object);
int z_bdeg_Point_fY(
               const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// bdeg_point.h
// …
namespace bdeg {
class Point {
  int d_x;
  int d_y;
 public:
  // CREATORS
  Point();
  Point(int x, int y);
  Point(const Point& other);
  ~Point()
  // MANIPULATORS
  Point& operator=(const Point& rhs);
  void setX(int x);
  void setY(int y);
  // ACCESSORS
  int x() const;
  int y() const;
};
// FREE OPERATORS
bool operator==(const Point& lhs,
                const Point& rhs);
// … (e.g., output operator)
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                     z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
        const z_bdeg_Point *object,
        const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

<- defensive programming

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <cassert>          <- defensive programming
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <bsls_assert.h>      <- defensive programming
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  assert(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  assert(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  assert(object);
  assert(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  assert(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  assert(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  assert(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  assert(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  assert(object);
  assert(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy(
                     z_bdeg_Point *object);
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg_Point *object,
                const z_bdeg_Point *other);
void z_bdeg_Point_fSetX(
              z_bdeg_Point *object, int x);
void z_bdeg_Point_fSetY(
              z_bdeg_Point *object, int y);
// ACCESSORS
int z_bdeg_Point_fX(
              const z_bdeg_Point *object);
int z_bdeg_Point_fY(
              const z_bdeg_Point *object);
// FREE OPERATIORS
int z_bdeg_Point_fIsEqual(
         const z_bdeg_Point *object,
         const z_bdeg_Point *other); // …
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <bsls_assert.h>        <- defensive programming
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate() {
  return new bdeg::Point;
}
z_bdeg_Point *z_bdeg_Point_fCreateInit(int x, int y) {
  return new bdeg::Point(x, y);
}
z_bdeg_Point *z_bdeg_Point_fCreateCopy(const z_bdeg_Point *other) {
  BSLS_ASSERT(other);
  return new bdeg::Point(*other);
}
void z_bdeg_Point_fDestroy(z_bdeg_Point *object) {
  BSLS_ASSERT(object);
  delete object;
}
// MANIPULATORS
void z_bdeg_Point_fAssign(z_bdeg_Point *object, z_bdeg_Point *other) {
  BSLS_ASSERT(object);
  BSLS_ASSERT(other);
  *object = *other;
}
void z_bdeg_Point_fSetX(z_bdeg_Point *object, int x) {
  BSLS_ASSERT(object);
  object->setX(x);
}
void z_bdeg_Point_fSetY(z_bdeg_Point *object, int y) {
  BSLS_ASSERT(object);
  object->setY(y);
}
// ACCESSORS
int z_bdeg_Point_fX(const z_bdeg_Point *object) {
  BSLS_ASSERT(object);
  return object->x();
}
int z_bdeg_Point_fY(const z_bdeg_Point *object) {
  BSLS_ASSERT(object);
  return object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  BSLS_ASSERT(object);
  BSLS_ASSERT(other);
  return *object == *other;
} // …
```

# Procedural Interface

```
// z_bdeg_point.h (continued)
// CREATORS
z_bdeg_Point *z_bdeg_Point_fCreate();
z_bdeg_Point *z_bdeg_Point_fCreateInit(
                                int x, int y);
z_bdeg_Point *z_bdeg_Point_fCreateCopy(
                const z_bdeg_Point *other);
void z_bdeg_Point_fDestroy
                      z_bdeg        *object)
// MANIPULATORS
void z_bdeg_Point_fAssign(
                      z_bdeg

void z_bdeg_Point
              z_bde
void z_bdeg_Point_fSe
              z_bdeg_Po
// ACCESSORS
int z_bdeg_

int z_bdeg_Point_
              const
// FREE OPERATIORS
int z_bdeg_Point
        const
        const z_bdeg_Poin
#ifdef __cplusplus
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <bsls_assert.h>        <- defensive programming
// CREATORS
z_bdeg_Point *z_b        int_fCreate() {
  return new bd        t;
}
z_bdeg_Poin        int_fCreateInit(int x, int y) {
  return        t(x, y);

        int_fCreate        const z_bdeg_Point *other) {
        nt(

        Point *object) {

        e, z_bdeg_Point *other) {

        ct, int x) {

        *object, int y) {

        st z_bdeg_Point *object) {
        object

        deg_Point_fY(const z_bdeg_Point *object) {
        ASSERT(object);
  re        n object->y();
}
// FREE OPERATORS
int z_bdeg_Point_fAreEqual(const z_bdeg_Point *object,
                           const z_bdeg_Point *other) {
  BSLS_ASSERT(object);
  BSLS_ASSERT(other);
  return *object == *other;
} // …
```

We must ensure

**exceptions**

don't leak out into

C code!

## 3. Total and Partial *Insulation* Techniques

# Procedural Interface

```
// z_bdeg_point.h (continued)
```

```
// z_bdeg_point.cpp
#include <z_bdeg_point.h>
#include <bdeg_point.h>
#include <bsls_assert.h>      <- defensive programming
```

**Clearly Architecturally Significant!**

```
} // extern "C" linkage
#endif
#endif // internal include guard
```

```
  BSLS_ASSERT(other);
  return *object == *other;
} // …
```

# Procedural Interface

# Discussion?

# Examples of Partial Insulation

1. `const` Array Access
2. Toy Stack
3. Adaptive Memory Pool (Simplified)

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{  2,  5, 11, 17, 23, 31,
  41, 47, 59, 67, 89, 101
};


const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{  2,  5, 11, 17, 23, 31,
  41, 47, 59, 67, 89, 101
};


const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{  2,  5, 11, 17, 23, 31,
  41, 47, 59, 67, 89, 101
};


const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{  2,  5, 11, 17, 23, 31,
  41, 47, 59, 67, 89, 101
};


const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{  2,  5, 11, 17, 23, 31,
  41, 47, 59, 67, 89, 101
};


const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{  2,  5, 11, 17, 23, 31,
  41, 47, 59, 67, 89, 101
};


const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{  2,  5, 11, 17, 23, 31,
  41, 47, 59, 67, 89, 101
};


const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

# `const` Array Access

```
// datautil.h
// …
class DataUtil {
  static const int *s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{   2,   7,  11,  17,  29,
   41,  53,  67,  97, 107,
  127, 139, 157, 191, 199
};

const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

Insulated Change!

# `const` Array Access

Client Does Not Recompile!

```
// datautil.h
// …
class DataUtil {
  static const int* s_data_p;
  static const int  s_length;

 public:
  static int data(int i) {
    return s_data_p[i];
  }

  static int dataLength() {
    return s_length;
  }
};
// …
```

Insulated Change!

```
// datautil.cpp
#include <datautil.h>
// …
static const int DATA[] =
{   2,   7,  11,  17,  29,
   41,  53,  67,  97, 107,
  127, 139, 157, 191, 199
};

const int *
DataUtil::s_data_p = DATA;

const int
DataUtil::s_length =
 sizeof DATA / sizeof *DATA;
```

## `const` Array Access

# Discussion?

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  Toy          *d_stack_p;
  std::size_t  d_capacity;
  std::size_t  d_length;
 private:
  void resize();
 public:
  Stack();
  // …
  void push(const Toy& v);
  // …
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  Toy          *d_stack_p;
  std::size_t  d_capacity;
  std::size_t  d_length;
 private:
  void resize();
 public:
  Stack();
  // …
  void push(const Toy& v);
  // …
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  Toy          *d_stack_p;
  std::size_t  d_capacity;
  std::size_t  d_length;
 private:
  void resize();
 public:
  Stack();
  // …
  void push(const Toy& v);
  // …
};
// …
```

```
// stack.cpp
#include <stack.h>
// …
Stack::Stack()
: d_stack_p(0)
, d_capacity(0)
, d_length(0)
{}
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  Toy          *d_stack_p;
  std::size_t  d_capacity;
  std::size_t  d_length;
 private:
  void resize();
 public:
  Stack();
  // …
  void push(const Toy& v);
  // …
};
// …
```

```
// stack.cpp
#include <stack.h>
// …
Stack::Stack()
: d_stack_p(0)
, d_capacity(0)
, d_length(0)
{}
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  Toy          *d_stack_p;
  std::size_t  d_capacity;
  std::size_t  d_length;
 private:
  void resize();
 public:
  Stack();
  // …
  void push(const Toy& v);
  // …
};
// …
```

```
// stack.cpp
#include <stack.h>
// …
Stack::Stack()
: d_stack_p(0)
, d_capacity(0)
, d_length(0)
{}
// …
void
Stack::push(const Toy& v) {
  if (d_length == d_capacity)
    resize();
  new(d_stack_p+d_length) Toy(v);
  ++d_length;
};
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  Toy          *d_stack_p;
  std::size_t  d_capacity;
  std::size_t  d_length;
 private:
  void resize();
 public:
  Stack();
  // …
  void push(const Toy& v);
  // …
};
// …
```

inline

```
// stack.cpp
#include <stack.h>
// …
Stack::Stack()
: d_stack_p(0)
, d_capacity(0)
, d_length(0)
{}
// …
void
Stack::push(const Toy& v) {
  if (d_length == d_capacity)
    resize();
  new(d_stack_p+d_length) Toy(v);
  ++d_length;
};
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

calling into .cpp file

```
// stack.cpp
#include <stack.h>

// … (Include any extra header files that
// … are needed.)

void Stack::resize() {
  std::size_t newCap =
    0 == d_capacity ? 1 : 2 * d_capacity;
  Toy *tmp = (Toy *)
    ::operator new(newCap * sizeof *tmp);
  NewDeleteProctor ndp(tmp);
  RangeProctor<Toy> rp(tmp);
  for (std::size_t i = 0;
         i < d_length; ++i, rp.advance())
    new(tmp+i) Toy(d_stack_p[i]);

  rp.release();  // Copy was successful.
  ndp.release();
  std::swap(d_stack_p, tmp); // +capacity
  d_capacity = newCap;
  for (std::size_t i = d_length; i;
                    (tmp + --i)->~Toy());
  ::operator delete(tmp);
}
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

calling into .cpp file

```
// stack.cpp
#include <stack.h>

// … (Include any extra header files that
// … are needed.)

void Stack::resize() {
  std::size_t newCap =
    0 == d_capacity ? 1 : 2 * d_capacity;
  Toy *tmp = (Toy *)
    ::operator new(newCap * sizeof *tmp);
  NewDeleteProctor ndp(tmp);
  RangeProctor<Toy> rp(tmp);
  for (std::size_t i = 0;
         i < d_length; ++i, rp.advance())
    new(tmp+i) Toy(d_stack_p[i]);

  rp.release();  // Copy was successful.
  ndp.release();
  std::swap(d_stack_p, tmp); // +capacity
  d_capacity = newCap;
  for (std::size_t i = d_length; i;
                    (tmp + --i)->~Toy());
  ::operator delete(tmp);
}
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

calling into .cpp file

```
// stack.cpp
#include <stack.h>

// … (Include any extra header files that
// … are needed.)

void Stack::resize() {
  std::size_t newCap =
    0 == d_capacity ? 2 : 1.5*d_capacity;
  Toy *tmp = (Toy *)
    ::operator new(newCap * sizeof *tmp);
  NewDeleteProctor ndp(tmp);
  RangeProctor<Toy> rp(tmp);
  for (std::size_t i = 0;
         i < d_length; ++i, rp.advance())
    new(tmp+i) Toy(d_stack_p[i]);

  rp.release();  // Copy was successful.
  ndp.release();
  std::swap(d_stack_p, tmp); // +capacity
  d_capacity = newCap;
  for (std::size_t i = d_length; i;
                    (tmp + --i)->~Toy());
  ::operator delete(tmp);
}
```

Insulated Change!

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

calling into .cpp file

```
// stack.cpp
#include <stack.h>

// … (Include any extra header files that
// … are needed.)

void Stack::resize() {
  std::size_t newCap =
    0 == d_capacity ? 2 : 1.5*d_capacity;
  Toy *tmp = (Toy *)
    ::operator new(newCap * sizeof *tmp);
  NewDeleteProctor ndp(tmp);
  RangeProctor<Toy> rp(tmp);
  for (std::size_t i = 0;
         i < d_length; ++i, rp.advance())
    new(tmp+i) Toy(d_stack_p[i]);

  rp.release();  // Copy was successful.
  ndp.release();
  std::swap(d_stack_p, tmp); // +capacity
  d_capacity = newCap;
  for (std::size_t i = d_length; i;
                    (tmp + --i)->~Toy());
  ::operator delete(tmp);
}
```

# Toy Stack

```
// stack.h
// …
#include <toy.h>
class Stack {
  // …
  void resize();
 public:
  // …
  void push(const Toy& v) {
    if (d_length == d_capacity)
      resize();
    new(d_stack_p+d_length) Toy(v);
    ++d_length;
  }
};
// …
```

calling into .cpp file

```
// stack.cpp
#include <stack.h>

// … (Include any extra header files that
// … are needed.)

void Stack::resize() {
  std::size_t newCap =
    0 == d_capacity ? 2 : 1.5*d_capacity;
  Toy *tmp = (Toy *)
    ::operator new(newCap * sizeof *tmp);
  NewDeleteProctor ndp(tmp);
  RangeProctor<Toy> rp(tmp);
  for (std::size_t i = 0;
        i < d_length, ++i, rp.advance())
    new(tmp+i) Toy(std::move(d_stack_p[i]));

  rp.release();  // Copy was successful.
  ndp.release();
  std::swap(d_stack_p, tmp);  // +capacity
  d_capacity = newCap;
  for (std::size_t i = d_length; --i;
                    (tmp
  ::operator delete(t
}
```

Insulated Optimization!

## Toy Stack

# Discussion?

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

```
class Multipool {
  // …
   void *allocate(size_t bytes);
   void  deallocate(void *addr);
};
```

# Adaptive Memory Pool (Simplified)

```
class Multipool {
  // …
   void *allocate(size_t bytes);
   void  deallocate(void *addr);
};
```

# Adaptive Memory Pool (Simplified)

```
class Multipool {
  // …
   void *allocate(size_t bytes);
   void  deallocate(void *addr);
};
```

# Adaptive Memory Pool (Simplified)

```
class Multipool {
  // …
   void *allocate(size_t bytes);
   void  deallocate(void *addr);
};
```

**Basis for General-Purpose Memory Allocator**

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

# Adaptive Memory Pool (Simplified)

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

# Adaptive Memory Pool (Simplified)

| | |
|---|---|
| BlockList | d_blockList; |
| size_t | d_blockSize; 56 |
| size_t | d_chunkSize; 32 |
| char | *d_begin_p; |
| char | *d_end_p; |
| Link | *d_freeList_p; |

# Adaptive Memory Pool (Simplified)

| | |
|---|---|
| BlockList | d_blockList; |
| size_t | d_blockSize; 56 |
| size_t | d_chunkSize; 32 |
| char | *d_begin_p; |
| char | *d_end_p; |
| Link | *d_freeList_p; |

Object Managing a Sequence of Memory Blocks of Arbitrary Size.

# Adaptive Memory Pool (Simplified)

| | |
|---|---|
| BlockList | d_blockList; |
| size_t | d_blockSize; 56 |
| size_t | d_chunkSize; 32 |
| char | *d_begin_p; |
| char | *d_end_p; |
| Link | *d_freeList_p; |

# Adaptive Memory Pool (Simplified)

| | |
|---|---|
| BlockList d_blockList; | |
| size_t d_blockSize; | 56 |
| size_t d_chunkSize; | 32 |
| char *d_begin_p; | |
| char *d_end_p; | |
| Link *d_freeList_p; | |

# Adaptive Memory Pool (Simplified)

| | |
|---|---|
| BlockList | d_blockList; |
| size_t | d_blockSize; 56 |
| size_t | d_chunkSize; 32 |
| char | *d_begin_p; |
| char | *d_end_p; |
| Link | *d_freeList_p; |

# Adaptive Memory Pool (Simplified)

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

# Adaptive Memory Pool (Simplified)

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

# Adaptive Memory Pool (Simplified)

| | |
|---|---|
| BlockList | d_blockList; |
| size_t | d_blockSize; 56 |
| size_t | d_chunkSize; 32 |
| char | *d_begin_p; |
| char | *d_end_p; |
| Link | *d_freeList_p; |

Next Available Block in Current Chunk

# Adaptive Memory Pool (Simplified)

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

# Adaptive Memory Pool (Simplified)

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

# Adaptive Memory Pool (Simplified)

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

# Adaptive Memory Pool (Simplified)

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

List of Individual Blocks Deallocated Since the Most-Recent Chunk was Allocated

# Adaptive Memory Pool (Simplified)

```
class Pool {

    //  ...

  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {

    //  ...

  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | ∘ |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 0 |
| char | *d_begin_p; | ∘ |
| char | *d_end_p; | ∘ |
| Link | *d_freeList_p; | ∘ |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 0 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | ∘ |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 0 |
| char | *d_begin_p; | ∘ |
| char | *d_end_p; | ∘ |
| Link | *d_freeList_p; | ∘ |

Empty List

Created with 56 as Constructor Argument

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 0 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

Empty List

Created with 56 as Constructor Argument

Arbitrary Sentinel Value (current chunk size)

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 1 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

Empty List

Created with 56 as Constructor Argument

Potentially Useful Sentinel Value (next chunk size)

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | ∘ |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 1 |
| char | *d_begin_p; | ∘ |
| char | *d_end_p; | ∘ |
| Link | *d_freeList_p; | ∘ |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | ∘ |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 1 |
| char | *d_begin_p; | ∘ |
| char | *d_end_p; | ∘ |
| Link | *d_freeList_p; | ∘ |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 2 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 2 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 4 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 8 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 8 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
class Pool {
```

| | | |
|---|---|---|
| BlockList | d_blockList; | |
| size_t | d_blockSize; | 56 |
| size_t | d_chunkSize; | 32 |
| char | *d_begin_p; | |
| char | *d_end_p; | |
| Link | *d_freeList_p; | |

```
  public:
    explicit Pool(size_t blockSize);
    void *allocate();
    void deallocate(void *blockAddress);
};
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued)
inline
Pool::Pool(size_t blockSize)
: d_blockList(blockSize)
, d_blockSize(blockSize)
, d_chunkSize(1) // special!
, d_begin_p(0)
, d_end_p(0)
, d_freeList_p(0) { }
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued)
inline
Pool::Pool(size_t blockSize)
: d_blockList(blockSize)
, d_blockSize(blockSize)
, d_chunkSize(1) // special!
, d_begin_p(0)
, d_end_p(0)
, d_freeList_p(0) { }
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued)
inline
Pool::Pool(size_t blockSize)
: d_blockList(blockSize)
, d_blockSize(blockSize)
, d_chunkSize(1) // special!
, d_begin_p(0)
, d_end_p(0)
, d_freeList_p(0) { }
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued)
inline
Pool::Pool(size_t blockSize)
: d_blockList(blockSize)
, d_blockSize(blockSize)
, d_chunkSize(1) // special!
, d_begin_p(0)
, d_end_p(0)
, d_freeList_p(0) { }
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued)
inline
Pool::Pool(size_t blockSize)
: d_blockList(blockSize)
, d_blockSize(blockSize)
, d_chunkSize(1) // special!
, d_begin_p(0)
, d_end_p(0)
, d_freeList_p(0) { }
inline void
Pool::deallocate(void *a) {
  static_cast<Link *>(a)->
    d_next_p = d_freeList_p;
  d_freeList_p =
     static_cast<Link *>(a);
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued)
inline
Pool
:
,
,
, d
d_end_p(0)
, d_freeList_p(0) { }
inline void
Pool::deallocate(void *a) {
  static_cast<Link *>(a)->
    d_next_p = d_freeList_p;
  d_freeList_p =
     static_cast<Link *>(a);
}
```

Link the block onto the front of the free list

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;
  void *getBlockFromNewChunk();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#inc
c

  si
  char        d_begin_p;
  char       *d_end_p;
  struct Link { d_next_p; }
I            *d_freeList_p;
  void *getBlockFromNewChunk();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

Is there a block available in the **current chunk**?

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#in
c

  si
  char        d_begi
  char       *d_end_p;
  struct Link { d_next_p;
I            *d_freeList_p;
  void *getBlockFromNewChunk();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

Is there a block available in the **free list**?

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#in
c

  si
  char        d_be      ;
  char       *d_end_
  struct Link { d_nex     }
I            *d_freeList_
  void *getBlockFromNewChun    ;
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

Allocate a new chunk; **return the first Block.**

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
I           *d_freeList_p;
  void *getBlockFromNewChunk();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 32 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 32 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 64 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 64 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {


  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 64 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {

  if (1 == d_chunksize) d_chunksize = 8;

  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::a
  void *p;
  if (d_begin_
    p = d_begin_p;
    d_begin_p +
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

More

Speed

Efficient?

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 64 }
co  t double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {

  if (1 == d_chunksize) d_chunksize = 8;

  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 32 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 32 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 16 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 16 }
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 16 }
const double k_GROWTH_FACTOR = 1.25;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 16 }
const double k_GROWTH_FACTOR = 1.25;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;

  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void *p;
  if (d_begin_p != d_end_p) {
    p = d_begin_p;
    d_begin_p += d_blockSize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 16 }
const double k_GROWTH_FACTOR = 1.25;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;

  if (d_chunksize < 4) ++d_chunksize;

  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::a
  void *p;
  if (d_begin_
    p = d_begin
    d_begin_p += d_blocksize;
  }
  else if (d_freeList_p) {
    p = d_freeList_p;
    d_freeList_p = p->d_next_p;
  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

More

Space

Efficient?

calling into .cpp file

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 16 }
co t double k_GROWTH_FACTOR = 1.25;

void *Pool::getBlockFromNewChunk() {
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;

  if (d_chunksize < 4) ++d_chunksize;

  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h (continued again)
inline
void *Pool::allocate() {
  void
  if
    p
    d
  }
  el

  }
  else
    p = getBlockFromNewChunk();
  return p;
}
```

```
// pool.cpp
#include <pool.h>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 16 }
       k_GROWTH_FACTOR = 1.25;

                                  ze;

                                  ze));

                                  e;

                                  ZE)
                                  ;

  return p;
}
```

**Partial Insulation Facilitates Performance Tuning**

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;

 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued again/revised)

inline
void *Pool::allocate() {
  if (d_begin_p == d_end_p)
    return getBlockDarnIt();
  void *p = d_begin_p;
  d_begin_p += d_blockSize;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chunkSize;
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued again/revised)

inline
void *Pool::allocate() {
  if (d_begin_p == d_end_p)
    return getBlockDarnIt();
  void *p = d_begin_p;
  d_begin_p += d_blockSize;
  return p;
}
```

Small!

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blockli                    lockList;
  size_t
  size_t
  char
  char
  struct

  void
 public
  Pool(size_
  void *allocate();
  void deallocate(void *a);
};
```

```
// pool.h (continued again/revised)

inline
void *Pool::allocate() {
  if (d_begin_p == d_end_p)
                    nIt();
                    ;
                    tSize;
```

**Partial Insulation Reduces Code Bloat!**

# Adaptive Memory Pool (Simplified)

```
// pool.h
#inc
cl

  char
  char        *d_end_p;
  struct Link { d_next_p; }
              *d_freeList_p;
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

If there is a block in the current chunk, return it ASAP!

```
// pool.h (continued again/revised)

inline
void *Pool::allocate() {
  if (d_begin_p == d_end_p)
    return getBlockDarnIt();
  void *p = d_begin_p;
  d_begin_p += d_blockSize;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#inc
cl

  char
  char        *d_end_p;
  struct Link { d_next_p; }
              *d_freeList_p;
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

If there isn't, whatever happens next is insulated!

```
// pool.h (continued again/revised)

inline
void *Pool::allocate() {
  if (d_begin_p == d_end_p)
    return getBlockDarnIt();
  void *p = d_begin_p;
  d_begin_p += d_blockSize;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_chun
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#include <blocklist.h>
class Pool {
  Blocklist  d_blockList;
  size_t     d_blockSize;
  size_t     d_
  char      *d_begin_p;
  char      *d_end_p;
  struct Link { d_next_p; }
            *d_freeList_p;
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

calling into .cpp file

```
// pool.cpp (revised)
#include <blocklist>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 32 };
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockDarnIt() {
  if (d_freeList_p) {
    Link *q = d_freeList_p;
    d_freeList_p = q->d_next_p;
    return q;
  }
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#inc
c

  si
  char        d_begin_p;
  char       *d_end_p;
  struct Link { d_next_p; }
             *d_freeList_p;
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

Is there a block available in the **free list**?

file

```
// pool.cpp (revised)
#include <blocklist>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 32 };
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockDarnIt() {
  if (d_freeList_p) {
    Link *q = d_freeList_p;
    d_freeList_p = q->d_next_p;
    return q;
  }
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

```
// pool.h
#inc
c

  si
  char        d_be
  char       *d_end_
  struct Link { d_next_   }
             *d_freeList_
  void *getBlockDarnIt();
 public:
  Pool(size_t blockSize);
  void *allocate();
  void deallocate(void *a);
};
```

Otherwise create a new chunk; **return its first block**

.cpp file

```
// pool.cpp (revised)
#include <blocklist>

// … (Include any extra header files that
// … are needed.)

enum { k_MAX_CHUNK_SIZE = 32 };
const double k_GROWTH_FACTOR = 2.0;

void *Pool::getBlockDarnIt() {
  if (d_freeList_p) {
    Link *q = d_freeList_p;
    d_freeList_p = q->d_next_p;
    return q;
  }
  size_t size = d_blockSize*d_chunkSize;
  char *p = static_cast<char *>
                  (::operator new(size));
  d_begin = p + d_blockSize;
  d_end   = p + size;
  d_chunkSize *= k_GROWTH_FACTOR;
  if (d_chunkSize > k_MAX_CHUNK_SIZE)
    d_chunkSize = k_MAX_CHUNK_SIZE;
  return p;
}
```

# Adaptive Memory Pool (Simplified)

Discussion?

# End of Section

# Questions?

# What Questions are we Answering?

- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What do we mean by *Total Insulation*?
- What are three general *Total Insulation* techniques?
- Which *Total-Insulation* technique is <u>not</u> Architectural?
- What do we mean by *Partial Insulation*?
- What are some examples of *Partial Insulation* techniques?
- What specific *benefits* do we get from insulation?
- What costs are generally associated with insulation?
- Why would one choose *Partial* over *Total* insulation?

# What Questions are we Answering?

- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What do we mean by *Total Insulation*?
- What are three general *Total Insulation* techniques?
- Which *Total-Insulation* technique is <u>not</u> Architectural?
- What do we mean by *Partial Insulation*?
- What are some examples of *Partial Insulation* techniques?
- What specific *benefits* do we get from insulation?
- What costs are generally associated with insulation?
- Why would one choose *Partial* over *Total* insulation?

# What Questions are we Answering?

- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What do we mean by *Total Insulation*?
- What are three general *Total Insulation* techniques?
- Which *Total-Insulation* technique is <u>not</u> Architectural?
- What do we mean by *Partial Insulation*?
- What are some examples of *Partial Insulation* techniques?
- What specific *benefits* do we get from insulation?
- What costs are generally associated with insulation?
- Why would one choose *Partial* over *Total* insulation?

# What Questions are we Answering?

- What do we mean by the term *Insulation*?
- How does *Insulation* compare with *Encapsulation*?
- Why/when would we put a `#include` directive in a `.h` file?
- What do we mean by *Total Insulation*?
- What are three general *Total Insulation* techniques?
- Which *Total-Insulation* technique is <u>not</u> Architectural?
- What do we mean by *Partial Insulation*?
- What are some examples of *Partial Insulation* techniques?
- What specific *benefits* do we get from insulation?
- What costs are generally associated with insulation?
- Why would one choose *Partial* over *Total* insulation?

# Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Survey of Advanced *Levelization* Techniques

   Avoiding Cyclic, Excessive, or Inappropriate Dependencies

3. Total and Partial *Insulation* Techniques

   Avoiding Unnecessary Compile-Time Coupling

4. Present-Day, Real-World Design Examples

   Prioritizing Physical Design to Improve Software Quality

# Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Survey of Advanced *Levelization* Techniques

   Avoiding Cyclic, Excessive, or Inappropriate Dependencies

3. Total and Partial *Insulation* Techniques

   Avoiding Unnecessary Compile-Time Coupling

4. Present-Day, Real-World Design Examples

   Prioritizing Physical Design to Improve Software Quality

# Introduction

All of the software we write is governed by a common overarching set of ***Organizing Principles***.

# Introduction

All of the software we write is governed by a common overarching set of

***Organizing Principles***.

Among the most central of which is achieving

***Sound Physical Design***.

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

## Where We Put Our Code Matters!

# Creating a Big Ball of Mud

## Where We Put Our Code Matters!

Each new edge is introduced by the addition of code that depends on code elsewhere.

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

# Creating a Big Ball of Mud

Over time …

# Creating a Big Ball of Mud

Over time …

# Large-Scale Physical Design

- Good physical design is an **engineering discipline**, **not** an afterthought.
- Good physical design must be introduced from the **inception** of an application.
- The physical design of our proprietary libraries should be **coherent** across the firm.

# Cyclic Link-time Dependency

# Cyclic Link-time Dependency

# Cyclic Link-time Dependency

# Cyclic Link-time Dependency

# Cyclic Link-time Dependency

# Excessive Link-time Dependency

# Classical Layered Architecture

# What Does `Account` Depend On?

```
class Account {
    // ...
  public:
    Account(int accountNumber);
        // Create an account
        // corresponding to the
        // specified 'accountNumber'
        // in the database.
    // ...
};
```

# On the Database!

# Everything Depends on the Database!

# Escalating Heavy-Weight Dependencies

# Breaking Dependencies Via Interfaces

# A Business Request

Suppose you are asked to provide some business functionality:

"Write me a 'Date' class that tells me whether today is a business day."

# What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day**."

# What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day**."

# What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day.**"

# What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day.**"

**Poor Logical Factoring**

# What's the Problem?

"Write me a 'Date' class that tells whether **today** is a **business day.**"

**NOT**

**Poor Logical Factoring**

**FLEXIBLE**

# What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day.**"

**Poor Logical Factoring**

**Poor Physical Design**

# What's the Problem?

"Write me a 'Date' class that tells me whether **today** is a **business day.**"

NOT MAINTAINABLE

Poor Logical Factoring

Poor Physical Design

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.
4. **<u>Provide well-factored useful components that we'll need over and over again!</u>**

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. **Represent a *date value* as a C++ Type.**
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.
4. **Provide well-factored useful components that we'll need over and over again!**

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Represent a *Date Value* as a C++ Type

# Solution 1: Represent a Date Value

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. **Determine what date value *today* is.**
3. Determine if a date value is a *business day*.
4. **Provide well-factored useful components that we'll need over and over again!**

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Determine what Date Value today is

# Solution 2: What Date is Today?

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. **Determine if a date value is a *business day*.**
4. **Provide well-factored useful components that we'll need over and over again!**

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Determine if a Date Value is a *Business Day*

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Wait a Minute: Where is the Data Source?

# Solution 3: Is `Date` a Business Day?

# The Original Request

"Write me a 'Date' class that tells me whether today is a business day."

What are the *real* requirements?

1. Represent a *date value* as a C++ Type.
2. Determine what date value *today* is.
3. Determine if a date value is a *business day*.
4. **Provide well-factored useful components that we'll need over and over again!**

# Non-Primitive Functionality

# Non-Primitive Functionality

# Non-Primitive Functionality

# Non-Primitive Functionality

# Non-Primitive Functionality

# Fine-Grained Reusable Class Design

# Rendering Software as Components

*Logical* content aggregated into a *Physical* hierarchy of ***components***

# Package Group Dependencies

# Client-Facing Component Diagram

# Client-Facing Component Diagram

# Client-Facing Component Diagram

# Client-Facing Component Diagram

# Client-Facing Component Diagram

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdet_date`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Implementing `bdecs_calendar`

# Hierarchically Reusable Implementation

## 4. Present-Day, Real-World Design Examples

# Hierarchically Reusable Implementation

## 4. Present-Day, Real-World Design Examples

# Hierarchically Reusable Implementation

# Foundation "Package-Group" Libraries

# End of Section

# Questions?

# Outline

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

2. Survey of Advanced *Levelization* Techniques

   Avoiding Cyclic, Excessive, or Inappropriate dependencies

3. Total and Partial *Insulation* Techniques

   Avoiding Unnecessary Compile-Time Coupling

4. Present-Day, Real-World Design Examples

   Prioritizing Physical Design to Improve Software Quality

# Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

# Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the <u>fundamental unit</u> of both *logical* and *physical* software design.

# Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the <u>fundamental unit</u> of both *logical* and *physical* software design.
- <u>**No**</u> *cyclic* dependencies/*long-distance* friendships.

# Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the <u>fundamental unit</u> of both *logical* and *physical* software design.
- <u>**No**</u> *cyclic* dependencies/*long-distance* friendships.
- Colocate logical constructs <u>only</u> with good reason: i.e., friendship; cycles; parts-of-whole; flea-on-elephant.

# Conclusion

1. Review of Elementary Physical Design

   Components, Modularity, Physical Dependencies

- A *Component* is the <u>fundamental unit</u> of both *logical* and *physical* software design.
- <u>**No**</u> *cyclic* dependencies/*long-distance* friendships.
- Colocate logical constructs <u>only</u> with good reason: i.e., friendship; cycles; parts-of-whole; flea-on-elephant.
- Put a `#include` in a header <u>only</u> with good reason: i.e., *Is-A*, *Has-A*, `inline`, `enum`, `typedef`-to-template.

# Conclusion

# The End