Introduction to C++

Part 2

Slides based on book:

Thinking in C++
by Bruce Eckel
Second edition
www.bruceeckel.com

FHA, SS 2005

Table of Contents

10 Name Control

- `static` storage
- Linkage
- Name spaces
- `static` members
- `static` member functions
- Linking C and C++ code

`static` 10

- Static storage:
  Object is created in a **`static`** data area rather than on the stack.
- Linkage:
  `static` controls the visibility of a name, i.e. what names the linker can see.

Example 10

```
// staticVariablesInFunctions.cpp
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char *charArray = 0) {
  static const char *s;   // retain value between
  if (charArray) {        //  function calls
    s = charArray;
    return *s;
  }
  else {
    require(s, "un-initialized s");
  }
  if (*s == '\0') return '\0';
  return *s++;
}                                    // cont'd...
```


Example cont'd 10

```
// ...cont'd

char *a = "abcdefghijklmnopqrstuvwxyz";


int main() {
  // oneChar();            // require(): fails
  oneChar(a);              // Initializes s to a
  char c;
  while ((c = oneChar()) != '\0') {
    cout << c << endl;
  }
}
```

Example 10

```
// staticObjectsInFunctions.cpp
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) { }            // Default
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);       // Only initialized once
  static X x2;           //   "       "       "
}

int main() {
  f();
}
```


`static` destructors I 10

- Destructors for **`static`** objects are called when leaving **`main()`** or calling **`exit()`**.
- Rule of thumb:
  Do not call **`exit()`** inside a constructor (you can end up with an infinite recursion).
- Remember: (**`static`**) destruction always occurs in the reverse order of initialization.

`static` destructors II 10

- Global objects are always constructed before **`main()`** is entered.
- Global objects are destroyed as **`main()`** exits.
- If a function containing a local **`static`** object is never called, the constructor is never executed - neither is the destructor.

Example 10

```
// staticDestructors.cpp
#include <fstream>
using namespace std;
ofstream out("statdest.out");     // Trace file
                                  // global static
class Obj {
  char c;                           // Identifier
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() for " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() for " << c << endl;
  }
};

Obj a('a');    // Global static storage
               // Constructor & destructor always
               // called                  cont'd...
```


Example cont'd 10

```
// ...cont'd
void f() {
  static Obj b('b');
}


void g() {
  static Obj c('c');
}


int main() {
  out << "inside main()" << endl;
  f();                    // Calls static con-
                          // structor for b
// g();   not called
  out << "leaving main()" << endl;
}
```


Output in *statdest.out* 10

```
Obj::Obj() for a
inside main()
Obj::Obj() for b
leaving main()
Obj::~Obj() for b
Obj::~Obj() for a
```

reverse order

Global constructors 10

- In C++, the constructor for a global **`static`** object is called, before **`main()`** is entered:
  A simple way to execute code before entering **`main()`**.
- In C only possible via startup code (written in assembly-language).

Linkage I 10

- All global objects implicitly have **`static`** storage:
  `int a = 0;`
  - **`a`** is stored in the **`static`** data area.
  - Name **`a`** is visible across all translation units (external linkage).
- Alternate way:
  `extern int a = 0;`

Linkage II 10

- All global **`static`** objects implicitly have **`static`** storage:
  `static int a = 0;`
  - **`a`** is stored in the **`static`** data area.
  - Name **`a`** is visible only in file (file scope).

Linkage III

10

- A variable in a function declared **`extern`** means that the storage exists elsewhere:

```
// f1.cpp
int main() {
  extern int i;
  std::cout << i;
}
```

```
// f2.cpp
int i = 5;
```


Linkage IV

10

- With functions **`static`** and **`extern`** only change visibility:
  `extern void f();`
  and
  `void f();`
  are globally visible.
- Visible in the file only (file scope):
  `static void f();`

Namespaces

10

- In big projects lack of control over the global name space can cause problems.
- In C++, the keyword **`namespace`** lets you subdivide the global name space.
- A **`namespace`** can only be defined on global scope, or nested within another **`namespace`**.

Example 10

```
// header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
  extern int x;
  void f();
}                 // No terminating ";" necessary
#endif
```

```
// header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "header1.h"
// Add more names to MyLib
namespace MyLib {          // NOT a redefinition!
  extern int y;
  void g();
}
#endif
```


Namespaces cont'd 10

- Each translation unit (file) contains one unnamed name space without an identifier.
- If you put local names in a unnamed name space, you don't need to give them internal linkage by making them **`static`**.
- C++ prefers unnamed name space over file **`static`**s.

Example 10

```
// unnamedNamespaces.cpp
namespace {
  class Arm { /* ... */ };
  class Leg { /* ... */ };
  class Head { /* ... */ };

  class Robot {
    Arm arm[4];
    Leg leg[16];
    Head head[3];
  } xanthan;

  int i, j, k;
}


int main() { }
```

The `using` directive I 10

- The **`using`** directive allows you to import an entire **`namespace`** at once. Names of this name space can then be used without qualification.
- **`using`** makes names appear as if they belong to the *nearest* enclosing **`namespace`** scope.

The `using` directive II 10

- Name spaces may be nested:

```
namespace Math {
  using namespace Int;
  Integer a, b;
  Integer divide(Integer, Integer);
}
```

- They can be used inside functions:

```
void arithmetic() {
  using namespace Int;
  Integer x;
  x.setSign(positive);
}
```


The `using` directive III 10

- There is more about **`using`** in connection with name spaces.
- Not covered here, but discussed in TiCpp.

`static` members 10

- The `static` member has to be declared within the class.
- The definition must occur outside the class.

Example 1 10

```
// statinit.cpp -- Scope of static initializer
#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
  }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x not ::x
```


Example 1 cont'd 10

```
int main() {
  WithStatic ws;
  ws.print();
}
```

Example 2 10

```
// staticArray.cpp
// Initializing static arrays in classes
class Values {
  // static consts are initialized in-place:
  static const int scSize = 100;
  static const long scLong = 100;
  // Automatic counting works with static arrays.
  // Arrays, non-integral and non-const statics
  // must be initialized externally:
  static const int scInts[];
  static const long scLongs[];
};

const int Values::scInts[] = {
  99, 47, 33, 11, 7
};

const long Values::scLongs[] = {
  99, 47, 33, 11, 7
};
```


Example 3 10

```
// staticObjectArrays.cpp
// Static arrays of class objects
class X {
  int i;
public:
  X(int ii) : i(ii) { }
};

class Stat {
  // Won't work:
// static const X x(100);
  // Both const and non-const static class
  // objects must be initialized externally:
  static X x2;
  static X xTable2[];
  static const X x3;
  static const X xTable3[];
};
```


Example 3 cont'd 10

```
X Stat::x2(100);

X Stat::xTable2[] = {
  X(1), X(2), X(3), X(4)
};

const X Stat::x3(100);

const X Stat::xTable3[] = {
  X(1), X(2), X(3), X(4)
};


int main() {
  Stat v;
}
```

`static` member functions 10

- **`static`** member functions work for a class as a whole rather than for an object of a class.
- A **`static`** member function can be called in association with an object.
- A more typical **`static`** member function call is by the function itself using the scope-resolution operator.

Example 10

```
// staticMemberFunctions.cpp
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
    // Non-static member function can access
    // static member function or data:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    // i++;     // Error: static member function
                // cannot access non-static member
                // data
    return ++j;
  }
                                        // cont'd...
```


Example cont'd 10

```
// ...cont'd

  static int f() {
    // val();   // Error: static member function
                // cannot access non-static member
                // function
    return incr();          // OK -- calls static
  }
};

int X::j = 0;


int main() {
  X x;
  X *xp = &x;
  x.f();
  xp->f();
  X::f();       // Only works with static members
}
```

Linking C++ and C code 10

- C and C++ compilers 'decorate' functions for the linker in different ways.

  Example:
  `float f(int a, char b);`
  is decorated into:
  `_f_int_char` (C++)
  `_f` (C)

Linking C++ and C code 10

- To give C decoration to **`f`** use:

```
extern "C" float f(int a, char b);
```

- For a group of declarations:

```
extern "C" {
  float f(int a, char b);
  double d(int a);
}
```

- For a header file:

```
extern "C" {
#include "MyHeader.h"
}
```


Summary 10

- The **`static`** keyword can be confusing as it has different meaning depending on the context.
- In C++ namespaces improve the name control, especially in large projects.
- The use of **`static`** (non-**`const`**) objects in classes is one more way to control names besides the concept of class variables.

Table of Contents

11 Refs & Copy-constructor

References
The copy-constructor
Pointer to members
Pointer to member functions

References in C++ 11

- A reference is like a **`const`**ant pointer that is automatically dereferenced.
- A reference must be initialized when it is created.
- Once a reference is initialized to an object, it cannot be changed to another object.
- There are no **`NULL`** references.

Example 11

```
// freeStandingReferences.cpp
#include <iostream>
using namespace std;

int y;
int& r = y;

const int& q = 12;

int x = 0;
int& a = x;


int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
}
```


References in functions 11

- The most common place to see references is as function arguments: Any modification made to the reference *inside* the function will cause a change to the reference *outside* this function.
- If returning a reference from a function be careful that this reference also exists outside the function scope.

Example 11

```
// reference.cpp -- simple C++ references
int* f(int* x) {
  (*x)++;
  return x;      // Safe, x is outside this scope
}

int& g(int& x) {
  x++;           // Same effect as in f()
  return x;      // Safe, outside this scope
}

int& h() {
  int q;
// return q;     // Error
  static int x;
  return x;      // Safe, x lives outside
                 // this scope
}
```

Example cont'd 11

```
int main() {
  int a = 0;
  f(&a);          // Ugly (but explicit)
  g(a);           // Clean (but hidden, you can't
                  // see that a is passed by
                  // reference)
}
```


`const` references 11

- If you know that a function will respect the `const`ness of an object, making the argument a `const` reference will allow the function to be used in all situations.
- Remember: temporary objects are always `const`ant.

Example 11

```
// constReferenceArguments.cpp
// Passing references as const

void f(int&) { }

void g(const int&) { }


int main() {
// f(1);              // Error
                      // Would make no sense
  g(1);
}
```

Arg-passing guidelines 11

- Your normal habit when passing args to a function should be to pass by **`const`** reference.
- Virtually the only time passing an address *is not* preferable is when an object would be 'damaged'. There the safe approach is to pass the object by value (thus copying it).

The copy-constructor 11

- The copy-constructor is used when passing objects *by value* into or out of a function.
- If you don't provide your own copy-constructor the compiler will automatically build a copy-constructor.
- This default copy-constructor copies bitwise (or memberwise if the object is more complex).

Copy-construction 11

- To prevent the compiler from doing a bitwise (or memberwise) copy you have to write your own copy-constructor.

Example 11

```
#include <string>
#include <iostream>
using namespace std;


class S {
  string str;
public:
  S();                    // No-arg constructor
  S(const S&);            // Copy-constructor
  S(const string&);       // One-arg constructor

  ~S() {
    cout << "Destructor. \"" << str << '"'
         << endl;
  }
};
```


Example cont'd 11

```
inline S::S() : str("") {
  cout << "No-arg-constructor" << endl;
}

inline S::S(const S&) : str("") {
  cout << "Copy-constructor" << endl;
}

inline S::S(const string& str) : str(str) {
  cout << "One-arg-constructor" << endl;
}

int main() {
  S s1;             // No-arg-constructor
  S s2("s2");
  s1 = s2;          // assigning
  S s3 = s1;        // Copy-constructor
}                   // Destructors
```


Output 11

```
No-arg-constructor
One-arg-constructor
Copy-constructor
Destructor. ""
Destructor. "s2"
Destructor. "s2"
```

Copy-construction 11

- The copy-constructor is often referred to as **`X(X&)`** ("X of X ref").
- **`S s2, s1; s1 = s2;`**
  and
  `S s2, s1 = s2;`
  are not the same as the previous example shows.

Composition 11

- More complex types represent types which build objects composed of other objects.
- The compiler creates a copy-constructor even if more complex types are involved.
- The compiler recursively calls the copy-constructors for all member objects.

Example 11

```
// defaultCopyConstructor.cpp
#include <iostream>
#include <string>
using namespace std;


class WithCC {             // With copy-constructor
public:
  WithCC() { }             // Required because of
                           // the copy-constructor
  WithCC(const WithCC&) {
    cout << "WithCC(WithCC&)" << endl;
  }
};
```

Example cont'd 11

```
class WoCC {           // Without copy-constructor
  string id;
public:
  WoCC(const string& ident = "") : id(ident) { }
  void print(const string& msg = "") const {
    if (msg.size() != 0) {
      cout << msg << ": ";
    }
    cout << id << endl;
  }
};
```


Example cont'd 11

```
class Composite {
  WithCC withcc;        // Embedded
  WoCC wocc;            // objects
public:
  Composite() : wocc("Composite()") { }
  void print(const string& msg = "") const {
    wocc.print(msg);
  }
};

int main() {
  Composite c;
  c.print("Contents of c");
  cout << "Calling Composite copy-constructor"
       << endl;
  Composite c2 = c;     // Calls copy-constructor
  c2.print("Contents of c2");
}
```


Output 11

```
Contents of c: Composite()
Calling Composite copy-constructor
WithCC(WithCC&)
Contents of c2: Composite()
```

Your own copy-constructor 11

- You need a copy-constructor only if you are passing an object of your class *by value*.
- If that never happens, you don't need a copy-constructor.

Bookkeeping of objects 11

- The next example shows the book-keeping of objects.
- Here you have to provide your own copy-constructor as default copying won't yield correct results.

Example 11

```
// HowMany2.cpp
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
  string name;              // Object identifier
  static int objectCount;
public:
  HowMany2(const string& id = "") : name(id) {
    ++objectCount;          // increment
    print("HowMany2()");
  }
  ~HowMany2() {
    --objectCount;          // decrement
    print("~HowMany2()");
  }
```

Example cont'd 11

```
  // The copy-constructor:
  HowMany2(const HowMany2& h) : name(h.name) {
    name += " copy";        // add " copy"
    ++objectCount;          // increment
    print("HowMany2(const HowMany2&)");
  }

  void print(const string& msg = "") const {
    if (msg.size() != 0) {
      out << msg << endl;
    }
    out << '\t' << name << ": "
        << "objectCount = "
        << objectCount << endl;
  }
};
```


Example cont'd 11

```
int HowMany2::objectCount = 0;

// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
  x.print("x argument inside f()");
  out << "Returning from f()" << endl;
  return x;
}

int main() {
  HowMany2 h("h");
  out << "Entering f()" << endl;
  HowMany2 h2 = f(h);
  h2.print("h2 after call to f()");
  out << "Call f(), no return value" << endl;
  f(h);
  out << "After call to f()" << endl;
}
```


HowMany2.out I 11

```
HowMany2()
    h: objectCount = 1
Entering f()
HowMany2(const HowMany2&)
    h copy: objectCount = 2
x argument inside f()
    h copy: objectCount = 2
Returning from f()
HowMany2(const HowMany2&)
    h copy copy: objectCount = 3
~HowMany2()
    h copy: objectCount = 2
h2 after call to f()
    h copy copy: objectCount = 2
```

HowMany2.out II 11

```
Call f(), no return value
HowMany2(const HowMany2&)
    h copy: objectCount = 3
x argument inside f()
    h copy: objectCount = 3
Returning from f()
HowMany2(const HowMany2&)
    h copy copy: objectCount = 4
~HowMany2()
    h copy: objectCount = 3
~HowMany2()
    h copy copy: objectCount = 2
After call to f()
~HowMany2()
    h copy copy: objectCount = 1
~HowMany2()
    h: objectCount = 0
```


Preventing pass-by-value 11

- You can prevent the compiler from creating a copy-constructor. You don't even need to create a definition.
- You prevent pass-by-value in declaring a **`private`** copy-constructor.

Example 11

```
// noCopyConstruction.cpp
// Preventing copy-construction

class NoCC {
  int i;
  NoCC(const NoCC&);       // No definition
public:
  NoCC(int ii = 0) : i(ii) { }
};

void f(NoCC);

int main() {
  NoCC n;
// f(n);         // Error: copy-constructor called
// NoCC n2 = n;  // Error: c-c called
// NoCC n3(n);   // Error: c-c called
}
```

Pointer to members 11

- A pointer to a member selects a location inside a class.
- Dilemma: There is no address inside a class.
- The syntax of pointers to members requires that you select an object at the same time you're dereferencing the pointer to the member.

Example 11

```
// ...
class Data {
public:
  int a, b, c;              // public!
  void print() const {
    cout << "a = " << a << ", b = " << b
         << ", c = " << c << endl;
  }
};

int main() {
  Data d, *dp = &d;
  int Data::*pmInt = &Data::a;
  dp->*pmInt = 47;          // Syntax:  ->*
  pmInt = &Data::b;
  d.*pmInt = 48;            // Syntax:  .*
  pmInt = &Data::c;
  dp->*pmInt = 49;
  dp->print();
}
```


Ptr to members functions 11

```
// pointerToMemberFunction.cpp
#include <iostream>
using namespace std;

class Widget {
public:
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
};


int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) const = &Widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
}
```

Summary 11

- C++ introduces references which are like `const`ant pointers that are automatically dereferenced by the compiler. You treat them like objects.
- The copy-constructor is used to create an new object from an existing one.
- Pointers-to-members behave in an ordinary way.

Table of Contents

12 Operator Overloading

Unary operators
Binary operators
Assignment operators
Arguments and return values
Non-member operators
Overloading assignment
Automatic type conversion
Reflexivity

Syntax I 12

- Defining an overloaded operator is like defining a function, but the name of the function is **`operator@`**, where `@` represents the operator being overloaded.
- Global unary operators require one argument.
- Global binary operators require two arguments.

Syntax II 12

- There is no argument for member unary operators.
- There is one argument for member binary operators.

Example 12

```
// operatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii) : i(ii) { }
  const Integer operator+(const Integer& rv) const {
    cout << "operator+" << endl;
    return Integer(i + rv.i);
  }
  Integer& operator+=(const Integer& rv) {
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};
```

Example cont'd 12

```
int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;

  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
}
```


Overloadable operators 12

- You can overload all existing unary operators.
- The same is true for existing binary operators.
- You can't overload:
  `operator.`
  `operator.*`
  user-defined operators like
  `operator**`

Unary operators 12

- Unary operators can be overloaded in the form of non-member **`friend`** functions, or
- They can be overloaded as member functions.
- The meaning of an operator depends on the way you want it to be used, but consider the client programmer's expectation.

Example 12

```
// overloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// Non-member functions:
class Integer {
  long i;
  Integer* This() { return this; }
public:
  Integer(long ll = 0) : i(ll) { }
  // No side effects takes const& argument:
  friend const Integer& operator+(const Integer& a);
  friend const Integer operator-(const Integer& a);
  // Prefix:
  friend const Integer& operator++(Integer& a);
  // Postfix:
  friend const Integer operator++(Integer& a, int);
  // ...
};
```


Example cont'd 12

```
// Prefix; return incremented value
const Integer& operator++(Integer& a) {
  cout << "++Integer\n";
  a.i++;
  return a;
}

// Postfix; return the value before increment:
const Integer operator++(Integer& a, int) {
  cout << "Integer++\n";
  Integer before(a.i);
  a.i++;
  return before;
}


// ... more in the book
```


Binary operators 12

- (Most) binary operators can be overloaded in the form of non-member **`friend`** functions, or
- They can be overloaded as member functions.
- The next example shows an excerpt of non-member and member functions (overloaded).
- More in the book.

Example 12

```
// Integer.h -- Overloaded operators
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

// Non-member functions:
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) { }
  // Operators that create new, modified value:
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
  friend const Integer
    operator-(const Integer& left,
              const Integer& right);
  // ...
};
#endif
```


Example cont'd 12

```
// Integer.cpp -- Overloaded operators
#include "Integer.h"
#include "../require.h"

const Integer operator+(const Integer& left,
                        const Integer& right) {
  return Integer(left.i + right.i);
}

const Integer operator-(const Integer& left,
                        const Integer& right) {
  return Integer(left.i - right.i);
}

// ... more in the book
```


Assignment operators 12

- All assignment operators should have code to check for self-assignment.
- The most important place to check for self-assignment is **`operator=`**.
- With complicated objects, **`a = a;`** may have desastrous results.
- **`a += a;`** may be acceptable.

Args and return values I 12

- In C++, the choice of arguments and return values follows a pattern it is wise to adapt.
- If you only read from an arg, pass it as **`const`** reference. Declare such member functions as **`const`**.
- **`operator+=`**, **`operator-=`** etc. which change the lvalue have no **`const`** left argument:

```
Integer& operator+=(
  Integer& left, const Integer& right);
```


Args and return values II 12

- The type of a return value depends on the expected meaning of the operator:

```
const Integer operator+(
  const Integer& left,
  const Integer& right);
```

- Assigning lets you also expect chained expressions like **`a = b = c;`** . **`const`** is ok, but this would prevent **`(a = b).func();`**.

Args and return values III 12

- For logical operators one expects to get at least an **`int`**, even better a **`bool`**.

`return` optimization 12

```
const Integer operator+(
   const Integer& left,
   const Integer& right)
{
   return Integer(left.i + right.i);
}
```

This is *not* an ordinary constructor call etc. Instead a temporary is built. The compiler then optimizes the passing out of the function.

`operator[]` 12

- The subscript operator must be a member function (with one argument).
- Because this operator implies that the object it's being called for acts like an array, you will return a reference from this operator: thus `obj[i]` can be used as an lvalue.

Non-member operators 12

- If it makes no difference realizing an operator as member function or as non-member operator choose the member option.
- If the left-hand is an operand of another type you have to choose the non-member realization. See next slide.

Example 12

```
// iostreamOperatorOverloading.cpp
// Example of non-member overloaded operators
#include "../require.h"
#include <iostream>
#include <sstream>    // "String streams"
#include <cstring>
using namespace std;

class IntArray {
  enum {sz = 5};      // enum hack seen in old code
  int i[sz];
public:
  IntArray() { memset(i, 0, sz * sizeof(*i)); }
  int& operator[](int x) {
    require(x >= 0 && x < sz,
      "IntArray::operator[] out of range");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os, const IntArray& ia);
```


Example cont'd I 12

```
  friend istream&
    operator>>(istream& is, IntArray& ia);
};

ostream& operator<<(ostream& os, const IntArray& ia)
{
  for (int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if (j != ia.sz - 1) {
      os << ", ";
    }
  }
  os << endl;
  return os;
}

istream& operator>>(istream& is, IntArray& ia){
  for (int j = 0; j < ia.sz; j++) {
    is >> ia.i[j];
  }
  return is;
}
```

Pass by reference

Example cont'd II 12

```
int main() {
  stringstream input("47 34 56 92 103");
  IntArray i;
  input >> i;
  i[4] = -1;          // Use overloaded operator[]
  cout << i;
}
```

Output

```
47, 34, 56, 92, -1
```

Overloading assignment 12

```
MyType b;          // 1)
MyType a = b;      // 2)
a = b;             // 3)
```

1) Constructor called
2) Copy-constructor called
3) **`MyType::operator=`** called

If the object hasn't been created yet, initialization is required; otherwise the assignment operator **`operator=`** is used.

Example 12

```
// copyingVsInitialization.cpp
class Fi {
public:
  Fi() { }
};

class Fee {
public:
  Fee(int) { }
  Fee(const Fi&) { }
};

int main() {
  Fee fee = 1;         // Fee(int)
  Fi fi;
  Fee fum = fi;        // Fee(Fi)

  Fee gee(1);          // Better style, same effect
  Fee gum(fi);
}
```


Behaviour of `operator=` 12

- The **`operator=`** has to be a member function: it is connected to the object to the left-hand side.
- All the necessary information has to be copied from the right-hand to the left-hand object.
- Be careful if there are any pointers in classes. Simply copying pointers means referring to the same storage location twice!

Example 12

```
// simpleAssignment.cpp -- Simple operator=()
#include <iostream>
using namespace std;

class Value {
  int a, b;
  float c;
public:
  Value(int aa = 0, int bb = 0, float cc = 0.0)
    : a(aa), b(bb), c(cc) { }

  Value& operator=(const Value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
};
```

Better check for self-assignment.
Not done here!

Pointers in classes 12

- Two approaches:
  1. Copy whatever the pointers refer to.
  2. Reference counting. The object that is pointed to knows how many objects are pointing to it. Constructors and destructors are responsible for proper bookkeeping.

Automatic `op=` creation 12

- The compiler automatically creates a **`Type::operator=(Type)`** if you don't make one yourself.
- With composed objects the assignment operator is called memberwise.

Example 12

```
// automaticOperatorEquals.cpp
#include <iostream>
using namespace std;

class Cargo {
public:
  Cargo& operator=(const Cargo&) {
    cout << "inside Cargo::operator=()" << endl;
    return *this;
  }
};

class Truck {
  Cargo b;
};

int main() {
  Truck a, b;
  a = b;     // Prints: "inside Cargo::operator=()"
}
```


Automatic type conversion 12

- Automatic type conversion is possible if the compiler finds a constructor for this type.
- This can be **`explicit`**ly prevented by the programmer.
- Automatic type conversion can also be produced through operator overloading (cp. example).

Example 1 12

```
// automaticTypeConversion.cpp
// Type conversion constructor
class One {
public:
  One() { }
};

class Two {
public:
  Two(const One&) { }
};

void f(Two) { }

int main() {
  One one;
  f(one);          // Wants a Two, has a One
}
```

Background activity:
`f(Two(one));`

Example 2


```
// operatorOverloadingConversion.cpp
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) { }
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) { }
  operator Three() const { return Three(x); }
};

void g(Three) { }

int main() {
  Four four(1);
  g(four);
  g(1);}
```

Background activity:
`g(Three(1, 0));`

Reflexivity

- Automatic type conversion may be applied to either operand (i.e. also the receiver operand).
- This is one of the most convenient reasons to use overloaded *global* operators.

Example


```
// reflexivityInOverloading.cpp
class Number {
  int i;
public:
  Number(int ii = 0) : i(ii) { }
  const Number
  operator+(const Number& n) const {
    return Number(i + n.i);
  }
  friend const Number
    operator-(const Number&, const Number&);
};

const Number             // Globally overloaded
  operator-(const Number& n1,
            const Number& n2) {
    return Number(n1.i - n2.i);
}
```

Example cont'd 12

```
int main() {
  Number a(47), b(11);
  a + b;    // OK
  a + 1;    // 2nd arg converted to Number
// 1 + a;   // Wrong! 1st arg not of type Number
  a - b;    // OK
  a - 1;    // 2nd arg converted to Number
  1 - a;    // 1st arg converted to Number
}
```


Summary 12

- Operator overloading is for the convenience of programmers and the readability of programs.
- With operator overloading the friends concept comes in handy (reflexivity).
- Some operators can't be over-loaded.
- C++ features automatic type conversion.

Table of Contents

13 Dynamic Object Creation

Object creation

`delete void *`

Running out of storage

(Overloading **`new`** and **`delete`**)

Object creation 13

- When a C++ object is created, two things occur:
  - Storage is allocated for the object
  - The constructor is called to initialize that storage.
- In C++, operator **`new`** takes care to create objects.
- **`delete`** calls the destructor and releases storage.

`delete void *` 13

- **`delete void *`** almost always is a bug in a program because type **`void`** has no destructor.
- Bug scenario:

```
void *b = new Obj(4, 'b');
delete b;
```

Cleanup responsibility 13

- To make **`Stash`** more flexible it will hold **`void`** pointers.
- This means that returned pointers must be cast to the proper type before deleting (or you'll get a memory leak).
- The user is responsible for cleaning up the objects.

`PStash` for pointers 13

```
// PStash.h -- Holds pointers instead of objects
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
  int quantity;     // Number of storage spaces
  int next;         // Next empty space
   // Pointer storage:
  void **storage;   // Array of pointers
  void inflate(int increase);
public:
  PStash() : quantity(0), storage(0), next(0) { }
  ~PStash();
  int add(void *element);
  void *operator[](int index) const;   // Fetch
  // Remove the reference from this PStash:
  void *remove(int index);
  // Number of elements in Stash:
  int count() const { return next; }
};
#endif
```


`PStash` cont'd 13

```
// PStash.cpp -- Pointer Stash definitions
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <cstring>        // 'mem' functions
using namespace std;

int PStash::add(void *element) {
  const int INFLATE_SIZE = 10;
  if (next >= quantity) { inflate(INFLATE_SIZE); }
  storage[next++] = element;
  return (next - 1);      // Index number
}

PStash::~PStash() {       // No ownership
  for (int i = 0; i < next; i++) {
    require(storage[i] == 0,
      "PStash not cleaned up");
  }
  delete []storage;
}
```

PStash cont'd


```
// Operator overloading replacement for fetch
void *PStash::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if (index >= next) {
    return 0;              // To indicate the end
  }
  // Produce pointer to desired element:
  return storage[index];
}


void *PStash::remove(int index) {
  void *v = operator[](index);
  // "Remove" the pointer:
  if (v != 0) {
    storage[index] = 0;
  }
  return v;
}
```


PStash cont'd


```
void PStash::inflate(int increase) {
  const int PSZ = sizeof(void *);
  void **st = new void *[quantity + increase];
  // Not strictly necessary:
  memset(st, 0, (quantity + increase) * PSZ);
  // Move existing data (probably faster than
  // looping):
  memcpy(st, storage, quantity * PSZ);
  // Bookkeeping
  quantity += increase;

  delete []storage;       // Old storage
  storage = st;           // Point to new memory
}
```


Testing **PStash** I


```
// PStashTest.cpp
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  PStash intStash;
  // 'new' works with built-in types, too. Note
  // the "pseudo-constructor" syntax:
  for (int i = 0; i < 25; i++) {
    intStash.add(new int(i));
  }
  for (int j = 0; j < intStash.count(); j++) {
    cout << "intStash[" << j << "] = "
         << *(int *)intStash[j] << endl;
  }
```

Testing `PStash` II 13

```
  // Clean up:
  for (int k = 0; k < intStash.count(); k++) {
    delete intStash.remove(k);
  }

  ifstream in ("PStashTest.cpp");
  assure(in, "PStashTest.cpp");
  PStash stringStash;
  string line;
  while (getline(in, line)) {
    stringStash.add(new string(line));
  }
  // Print out the strings:
  for (int u = 0; stringStash[u]; u++) {
    cout << "stringStash[" << u << "] = "
         << *(string *)stringStash[u] << endl;
  }
  // Clean up:
  for (int v = 0; v < stringStash.count(); v++) {
    delete (string *)stringStash.remove(v);
  }
}
```


Running out of storage 13

- If you run out of storage a function called *new-handler* is invoked.
- The default behaviour for the *new-handler* is to throw an exception.
- It is wise to at least replace the *new-handler* with a message to tell you if you run out of memory.
- You replace the new-handler by including **`<new>`**.

Example 13

```
// newHandler.cpp -- Changing the new-handler
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int count = 0;

void out_of_memory() {
  cerr << "memory exhausted after " << count
       << " allocations!" << endl;
  exit(1);
}

int main() {
  set_new_handler(out_of_memory);
  while(1) {
    count++;
    new int[1000];      // Exhausts memory
  }
}
```

new-handler

Overloading `new` & `delete` 13

- Covered in the book (not introduced here):
  - Overloading `operator new()`
  - Overloading `operator delete()`
  - Overloading globally or for a class
  - Overloading for arrays
  - Placing an object in a specific location/explicit destructor call

Summary 13

- To solve general programming problems you must be able to create and destroy objects at any time.
- `new` and `delete` allow easy dynamic object creation/destruction.
- The behaviour of `new` and `delete` can be changed to fit your needs.
- You can influence what happens when the heap runs out of storage.

Table of Contents

14 Inheritance & Composition

Code reuse
Inheritance
Composition and Inheritance
Name hiding
Upcasting
Early binding

Code reuse I 14

- Code is reused by creating new classes.
- Instead of creating new classes from scratch existing classes are used someone else has built and debugged.
- The trick is to use classes without changing code of those classes.

Code reuse II 14

- There are two approaches for code reuse:
- Composition: you create objects composed of members of existing classes.
- Inheritance: you create a new class as a type of an existing class.

Inheritance syntax 14

- A class that inherits from a base class is defined as follows:

```
class NewClass :
  public BaseClass { };
```

- If inheriting from multiple classes the comma separator is used.

Inheritance 14

- The new class inherits all data elements and all member functions from its base class.
- All the **`private`** members are still **`private`** in the subclass.
- During inheritance all derived classes default to **`private`** unless the base class is preceded by the **`public`** keyword.

Example 14

```
// useful.h
// A class to reuse
#ifndef USEFUL_H
#define USEFUL_H


class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};

#endif
```

Example cont'd 14

```
// inheritance.cpp
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
  int i;                // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute();      // Different name call
    return i;
  }
  void set(int ii) {
    i = ii;
    X::set(ii);         // Same-name function call
  }
};
```


Example 14

```
int main() {
  // sizeof(Y) is twice as big as sizeof(X):
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = " << sizeof(Y) << endl;

  Y d;
  d.change();

  // X function interface comes through:
  d.read();
  d.permute();

  // Redefined functions hide base versions:
  d.set(12);
}
```


Constructor initializer list 14

- When an object is created the compiler guarantees that constructors for all of its subobjects are called.
- **`private`** members of the superclass can only be initialized in calling a constructor in the constructor initializer list.

Composition & inheritance 14

- Composition and inheritance are often used together:

```
// combined.cpp -- Inheritance & composition
class A {
  int i;
public:
  A(int ii) : i(ii) { }
  ~A() { }
  void f() const { }
};

class B {
  int i;
public:
  B(int ii) : i(ii) { }
  ~B() { }
  void f() const { }
};                                  // cont'd...
```


Example cont'd 14

```
class C : public B {
  A a;
public:
  C(int ii) : B(ii), a(ii) { }

  ~C() { }               // Calls ~A() and ~B()

  void f() const {       // Redefinition
    a.f();
    B::f();
  }
};


int main() {
  C c(47);
}
```


Automatic destructor calls 14

- You never need to make explicit destructor calls because there is only one destructor for any class.
- The compiler ensures that all destructors are called, starting from the most recently derived destructor up to the base-class destructor.

Name hiding 14

- If you rewrite a base-class function name with the same signature in a derived class, this is called
  - *redefining* for an ordinary member function and
  - *overwriting* for a **`virtual`** member function.
- Considered the same signature(!):
  `Base::f(void *)`
  `Sub::f(string *)`

Example 14

```
// inheritStack.cpp -- Specializing the Stack class
#include "Stack4.h"       // all functions inline
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {
public:
  // Hides Stack::push(void *):
  void push(string *str) { Stack::push(str); }
  string *peek() const {
    return (string *)Stack::peek();
  }
  string *pop() {
    return (string *)Stack::pop();
  }
```


Example cont'd 14

```
  ~StringStack() {
    string *top = pop();
    while(top) { delete top; top = pop(); }
  }
};

int main() {
  ifstream in("inheritStack.cpp");
  assure(in, "inheritStack.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string *s;
  while((s = textlines.pop()) != 0) { // No cast
    cout << *s << endl;
    delete s;
  }
}
```

Op overloading/inheritance 14

- Except for the assignment operator **`operator=`**, operators are automatically inherited into the derived class.

Multiple inheritance 14

- You can inherit from multiple classes, yet you should not try this until you are familiar with C++.
- By that time, you'll probably realize that single inheritance in most cases will do.

Upcasting I 14

- Upcasting means casting from the derived class up to the base class.
- Upcasting is always safe because you are moving from a specialized class to a more general one.
- This is why the compiler allows upcasting without an explicit cast.

Upcasting II

- Upcasting can also occur during a simple assignment to a pointer or a reference:

```
Wind w;
Instrument *ip; // Instrument is
                // base class of
                // Wind
ip = &w;              // Upcast
Instrument& ir = w;   // Upcast
```


Upcasting III

- Any upcast loses type information about an object.
- Information can't be regained (because of early binding) - unless a function is implemented as a **`virtual`** function (late binding).

Summary

- New types can be built with inheritance and composition.
- Composition is used to reuse existing types as part of the underlying implementation of the new type.
- Inheritance is used to force the new type to be the same type as the base class (plus maybe added state and functionality).

Table of Contents

15 Polymorphism

Early / late binding
`virtual` functions
Abstract base classes
Pure **`virtual`** functions
Pure **`virtual`** definitions
`virtual` constructors etc.
Downcasting

An essential feature 15

- Polymorphism (in C++ implemented with **`virtual`** functions) is the third essential feature of an object-oriented language, after data abstraction and inheritance.
- Polymorphism decouples the *what* from the *how*.

Function call binding 15

- Connecting a function call to a function body is called *binding*.
- If binding is performed during compile-time (compiler/linker) this is called *early binding*.
- If binding occurs at run-time this is called *late* or *dynamic binding*.
- Late binding asks for a mechanism to determine the type of an object at run-time.

`virtual` functions 15

- To cause late binding to occur for a particular function, C++ uses the **`virtual`** keyword.
- If a function is declared **`virtual`** in the base class, it is **`virtual`** in all the derived classes.
- You normally declare a function **`virtual`** only in the base class. You omit **`virtual`** when overriding such a function in a subclass.

Example 15

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat };    // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};
```

Example cont'd 15

```
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute);              // Upcasting
}
```


Abstract base classes 15

- You make a class *abstract* if you want to provide a base class only to upcast to so that its interface can be used.
- A class is abstract if it has at least one *pure* ***`virtual`*** *function*.
- Syntax of a pure **`virtual`** function:
  `virtual void f() = 0;`

Example 15

```
// Instrument5.cpp -- Pure abstract base classes
#include <iostream>
using namespace std;

enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char *what() const = 0;

  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
```

Example cont'd 15

```
class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char *what() const { return "Wind"; }
  void adjust(int) { }
};


class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char *what() const { return "Percussion"; }
  void adjust(int) { }
};
```


Example cont'd 15

```
class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char *what() const { return "Stringed"; }
  void adjust(int) { }
};


class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char *what() const { return "Brass"; }
};
```


Example cont'd 15

```
class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char *what() const { return "Woodwind"; }
};


// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }
```

Example cont'd 15

```
int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;

  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);

  f(flugelhorn);
}
```


Pure **`virtual`** definitions 15

- You can provide a *definition* for a pure **`virtual`** function in the base class.
- You're still telling the compiler not to allow objects of that class, and the pure **`virtual`** functions must still be defined in a derived class in order to create objects.

Example 15

```
// pureVirtualDefinitions.cpp
// Pure virtual base definitions
#include <iostream>
using namespace std;

class Pet {
public:
  virtual void speak() const = 0;
  virtual void eat() const = 0;

  // Inline pure virtual definitions illegal:
  // virtual void sleep() const = 0 { }
};

// OK, not defined inline. Pure virtual definition:
void Pet::eat() const {
  cout << "Pet::eat()" << endl;
}
```

Example cont'd 15

```
void Pet::speak() const {
  cout << "Pet::speak()" << endl;
}


class Dog : public Pet {
public:
  // Use the common Pet code:
  void speak() const { Pet::speak(); }
  void eat() const { Pet::eat(); }
};


int main() {
  Dog simba;              // Richard's dog
  simba.speak();
  simba.eat();
}
```


`virtual`s in constructors 15

- A **`virtual`** function called inside a constructor *always* binds the local version of that function.
  In other words: the **`virtual`** mechanism does *not* work within constructors.
- Many compilers will even perform early binding as late binding would only produce a call to the local function.

`virtual` destructors 15

- You cannot use the **`virtual`** keyword with constructors.
- Destructors however can and often must be **`virtual`**.
- Destructors must be **`virtual`** if you want to manipulate an object through a pointer to its base class (i.e. through its generic interface). Cp. next example.

Example 15

```
// virtualDestructors.cpp
// Behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};
```


Example cont'd 15

```
class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1;    // Upcast
  delete bp;
  Base2* b2p = new Derived2;   // Upcast
  delete b2p;
}
```

Output

```
~Base1()
~Derived2()
~Base2()
```


Pure `virtual` destructors 15

- 'Pure' **`virtual`** destructors must have a code body.
- A pure **`virtual`** destructor causes its class to be abstract.
- When inheriting from a class that contains a pure **`virtual`** destructor, you are not required to privide a definition of a pure **`virtual`** destructor in the derived class.

Example 1 15

```
// unAbstract.cpp
// Pure virtual destructors seem to behave
// strangely

class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() { }

class Derived : public AbstractBase { };
// class Derived not abstract as the pure virtual
// destructor of the base class is automatically
// overridden in Derived.

int main() {
  Derived d;
}
```


Next example 15

```
// pureVirtualDestructors.cpp
// Pure virtual destructors require a
// function body
#include <iostream>
using namespace std;


class Pet {
public:
  virtual ~Pet() = 0;
};


Pet::~Pet() {
  cout << "~Pet()" << endl;
}
```


Next example cont'd 15

```
class Dog : public Pet {
public:
  ~Dog() {
    cout << "~Dog()" << endl;
  }
};


int main() {
  Pet *p = new Dog;    // Upcast
  delete p;            // Virtual destructor call
}
```

`virtual`s in destructors 15

- Inside a destructor only the 'local' version of a member function is called even if this member is **`virtual`**:
  The **`virtual`** mechanism is ignored.

Example 15

```
// virtualsInDestructors.cpp
// Virtual calls inside destructors
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "~Base()\n";
    f();
  }
  virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};
```


Example cont'd 15

```
int main() {
  Base* bp = new Derived;      // Upcast
  delete bp;
}
```

Output

```
~Derived()
~Base()
Base::f()
```

`virtual` operators 15

- Operators can be made `virtual` just like other member functions.
- Implementing `virtual` operators often becomes confusing because you may be operating on two objects, both with unknown types.

Example 15

```
// operatorPolymorphism.cpp
// Polymorphism with overloaded operators
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix *) = 0;
  virtual Math& multiply(Scalar *) = 0;
  virtual Math& multiply(Vector *) = 0;
  virtual ~Math() { }
};
```


Example cont'd 15

```
class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix *) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
  Math& multiply(Scalar *) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector *) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};
```

Example cont'd 15

```
class Scalar : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix *) {
    cout << "Matrix * Scalar" << endl;
    return *this;
  }
  Math& multiply(Scalar *) {
    cout << "Scalar * Scalar" << endl;
    return *this;
  }
  Math& multiply(Vector *) {
    cout << "Vector * Scalar" << endl;
    return *this;
  }
};
```


Example cont'd 15

```
class Vector : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix *) {
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar *) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector *) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};
```


Example cont'd 15

```
int main() {
  Matrix m;
  Vector v;
  Scalar s;

  // Multiply any two Math objects:
  Math *math[] = { &m, &v, &s };
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      Math& m1 = *math[i];
      Math& m2 = *math[j];
      // Two upcast Math references:
      // Use multiple dispatching:
      m1 * m2;
    }
  }
}
```

Downcasting 15

- C++ provides a special explicit cast which is a type-safe downcast operation: **`dynamic_cast`**
- If incorrect you get **`NULL`** back.
- Use **`dynamic_cast`** only with **`virtual`** functions.

Example 15

```
// dynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){ } };
class Dog : public Pet { };
class Cat : public Pet { };

int main() {
  Pet *b = new Cat;           // Upcast
  // Try to cast it to Dog *:
  Dog *d1 = dynamic_cast<Dog *>(b);
  // Try to cast it to Cat *:
  Cat *d2 = dynamic_cast<Cat *>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
}
```


Summary 15

- In C++ polymorphism (= dynamic binding) is implemented with **`virtual`** functions.
- Polymorphism is not possible without using data abstraction and inheritance.
- Polymorphism requires to view commonalities among classes. The benefit is faster development, better code and easier maintenance.

Table of Contents

16 Templates

Syntax
Constants in templates
Iterators

`template` solution 16

- A **`template`** represents a way to reuse code, mostly in connection with containers.
- It is based on code-substitution performed by the compiler.
- Instead of reusing object code as with inheritance and composition it reuses *source code*.
- A **`template`** implements the concept of *parameterized type*.

`template` syntax 16

- The `template` keyword tells the compiler that the class definition that follows will manipulate one or more unspecified types.
- At code-generation time the types must be known so that the compiler can substitute them.

Example 16

```
// Array.cpp
// Produce a bounds-checked array

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

template<class T> class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return A[index];
  }
};
```

Substitution parameter

Example cont'd 16

```
int main() {
  // Array of type int:
  Array<int> ia;
  // Array of type float:
  Array<float> fa;
  for (int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }

  for (int j = 0; j < 20; j++) {
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
  }
}
```

Non-inline function defs 16

- In cases of non-**`inline`** member function definitions the compiler needs to see the **`template`** declaration before the member function definition.

Example 16

```
// Array2.cpp -- Non-inline template definition
#include "../require.h"

template<class T> class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
  require(index >= 0 && index < size,
    "Index out of range");
  return A[index];
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
}
```


`IntStack` as a **`template`** 16

- A **`template`** makes some assumptions about the objects it is holding.
- **`template`**s represent a kind of *weak typing* mechanism in C++, which is ordinarily a strongly-typed language.

StackTemplate 16

```
// StackTemplate.h -- Simple stack template
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T> class StackTemplate {
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) { }
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  int size() { return top; }
};
#endif
```


Test example 16

```
// StackTemplateTest.cpp
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for (int i = 0; i < 20; i++) {
    is.push(fibonacci(i));
  }
  for (int k = 0; k < 20; k++) {
    cout << is.pop() << endl;
  }
  ifstream in("StackTemplateTest.cpp");
  assure(in, "StackTemplateTest.cpp");
  string line;
```


Test example cont'd 16

```
  StackTemplate<string> strings;
  while(getline(in, line)) {
    strings.push(line);
  }

  while(strings.size() > 0) {
    cout << strings.pop() << endl;
  }
}
```

Constants in `templates` 16

- **`template`** arguments are not restricted to class types; you can use built-in types.
- Values of built-in types become compile-time constants.
- Default values for these arguments can be used as well.

Example 16

```
// Array3.cpp
// Built-in types as template arguments
#include "../require.h"
#include <iostream>
using namespace std;


template<class T, int size = 100> class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return array[index];
  }

  int length() const { return size; }
};
```


Example cont'd 16

```
class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) { }

  Number& operator=(const Number& N) {
    f = N.f;
    return *this;
  }

  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& X) {
      return os << X.f;
  }
};
```

Example cont'd 16

```
template<class T, int size = 20> class Holder {
  Array<T, size> *np;
public:
  Holder() : np(0) { }
  T& operator[](int i) {
    require(0 <= i && i < size);
    if (!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};


int main() {
  Holder<Number> h;
  for (int i = 0; i < 20; i++) { h[i] = i; }
  for (int j = 0; j < 20; j++) {
    cout << h[j] << endl;
  }
}
```


Templatized **`PStash`** 16

- Reorganizing **`PStash`** into a **`template`** requires to change a number of functions to be non-**`inline`**.
- As a **`template`** these functions still belong into the header file.
- The increment used by **`inflate()`** has been templatized. It can be initialized at the point of instantiation.

Example 16

```
// TPStash.h
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10> class PStash {
  int quantity;    // Number of storage spaces
  int next;        // Next empty space
  T **storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), next(0), storage(0) { }
  ~PStash();
  int add(T *element);
  T *operator[](int index) const;    // Fetch
  // Remove the reference from this PStash:
  T *remove(int index);
  // Number of elements in Stash:
  int count() const { return next; }
};
```

Example cont'd 16

```
template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if (next >= quantity) { inflate(incr); }
  storage[next++] = element;
  return(next - 1);        // Index number
}


// Ownership of remaining pointers:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for (int i = 0; i < next; i++) {
    delete storage[i];     // Null pointers OK
    storage[i] = 0;        // Just to be safe
  }
  delete []storage;
}
```


Example cont'd 16

```
template<class T, int incr>
T *PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if (index >= next) { return 0; }
  require(storage[index] != 0,
    "PStash::operator[] returned null pointer");
  // Produce pointer to desired element:
  return storage[index];
}


template<class T, int incr>
T *PStash<T, incr>::remove(int index) {
  // operator[] performs validity checks:
  T *v = operator[](index);
  // "Remove" the pointer:
  if(v != 0) { storage[index] = 0; }
  return v;
}
```


Example cont'd 16

```
template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int PSZ = sizeof(T *);
  T **st = new T *[quantity + increase];
  memset(st, 0, (quantity + increase) * PSZ);
  memcpy(st, storage, quantity * PSZ);
  quantity += increase;
  delete []storage;        // Old storage
  storage = st;            // Point to new memory
}

#endif // TPSTASH_H
```

Iterators 16

- An iterator is an object that moves through a container of objects and selects one at a time.
- There is no direct access to the implementation of the container.
- Iterators provide a standard way to access elements.
- An iterator is also a design pattern.

Example 16

```
// IterIntStack.cpp
// Simple integer stack with iterators
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;


class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) { }
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
```


Example cont'd 16

```
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  friend class IntStackIter;
};


// cont'd...
```

Example cont'd 16

```
// An iterator is like a "smart" pointer:
class IntStackIter {
  IntStack& s;
  int index;
public:
  IntStackIter(IntStack& is) : s(is), index(0) { }
  int operator++() {           // Prefix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[++index];
  }
  int operator++(int) {        // Postfix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[index++];
  }
};
```


Example cont'd 16

```
int main() {
  IntStack is;
  for (int i = 0; i < 20; i++) {
    is.push(fibonacci(i));
  }

  // Traverse with an iterator (without popping):
  IntStackIter it(is);
  for (int j = 0; j < 20; j++) {
    cout << it++ << endl;
  }
}
```


Avoiding name clashes 16

- To avoid name clashes containers can be built with an associated iterator class nested into the container.
- This is what the Standard C++ Library does.
- See example above reworked with a nested iterator in the book.

Using iterators of the STL 16

- When using **`template`** classes (containers) the STL provides iterators ready to use.

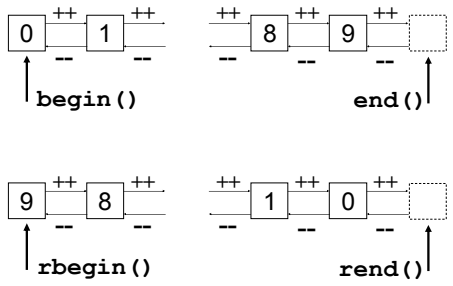
Example 16

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
  vector<int> myVector(10);

  for (int cnt = 0; count < 10; ++ cnt) {
    myVector[cnt] = cnt;
  }
 // Initialize a forward iterator:
  vector<int>::iterator fit = myVector.begin();
  // Initialize a backward iterator:
  vector<int>::reverse_iterator bit =
    myVector.rbegin();
```


Example cont'd 16

```
  // An iterator is like a "smart" pointer:
  cout << "Iterating myVector:" << endl;
  while (fit != myVector.end()) {
    cout << *fit++ << " ";
  }
  cout << endl;

  cout << "Reverse-iterating myVector:" << endl;
  while (bit != myVector.rend()) {
    cout << *bit++ << " ";
  }
  cout << endl;
}
```

Navigation 16



Summary 16

- Container classes are an essential part of object oriented programming.
- Templates help to easily use containers.
- Containers and iterators are in the Standard C++ Library and thus available with every compiler.
- Containers virtually fulfill all your needs. You normally don't have to create your own ones.