The C++ Language

COMS W4995-02

Prof. Stephen A. Edwards
Fall 2002
Columbia University
Department of Computer Science

The C++ Language

Bjarne Stroupstrup, the language's creator, explains

C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming.

C++ Features

Classes
  User-defined types

Operator overloading
  Attach different meaning to expressions such as a + b

References
  Pass-by-reference function arguments

Virtual Functions
  Dispatched depending on type at run time

Templates
  Macro-like polymorphism for containers (e.g., arrays)

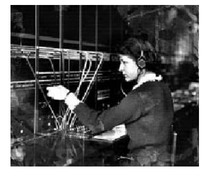
Exceptions
  More elegant error handling

Implementing Classes

Simple without virtual functions.

C++	Equivalent C
`class Stack {`	`struct Stack {`
`  char s[SIZE];`	`  char s[SIZE];`
`  int sp;`	`  int sp;`
`public:`	`};`
`  Stack();`	
`  void push(char);`	`void St_Stack(Stack*);`
`  char pop();`	`void St_push(Stack*,char);`
`};`	`char St_pop(Stack*);`

Operator Overloading



For manipulating user-defined "numeric" types

```
complex c1(1, 5.3), c2(5); // Create objects

complex c3 = c1 + c2; // + means complex plus

c3 = c3 + 2.3; // 2.3 promoted to a complex number
```

Complex Number Type

```
class Complex {
  double re, im;
public:
  complex(double); // used, e.g., in c1 + 2.3
  complex(double, double);

  // Here, & means pass-by-reference: reduces copying
  complex& operator+=(const complex&);
};
```

References

Designed to avoid copying in overloaded operators

Especially efficient when code is inlined.

A mechanism for calling functions pass-by-reference

C only has pass-by-value: fakable with explicit pointer use

```
void bad_swap(int x, int y) {
  int tmp = x; x = y; y = tmp;
}

void swap(int &x, int &y) {
  int tmp = x; x = y; y = tmp;
}
```

Function Overloading

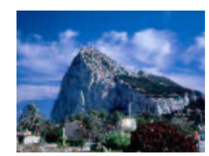
Overloaded operators a particular case of function/method overloading

General: select specific method/operator based on name, number, and type of arguments.

Return type not part of overloading

```
void foo(int);
void foo(int, int); // OK
void foo(char *);   // OK
int  foo(char *);   // BAD
```

Const



Access control over variables, arguments, and objects.

```
const double pi = 3.14159265; // Compile-time constant

int foo(const char* a) { // Constant argument
  *a = 'a';              // Illegal: a is const
}

class bar {
  // "object not modified"
  int get_field() const { return field; }
};
```

Templates

Macro-preprocessor-like way of providing polymorphism.

Polymorphism: Using the same code for different types

Mostly intended for containiner classes (vectors of integers, doubles, etc.)

Standard Template Library has templates for strings, lists, vectors, hash tables, trees, etc.

Template Stack Class

```
template <class T> class Stack {
  T s[SIZE]; // T is a type argument
  int sp;
public:
  Stack() { sp = 0; }
  void push(T v) {
    if (sp == SIZE) error("overflow");
    s[sp++] = v;
  }
  T pop() {
    if (sp == () error("underflow");
    return s[--sp];
  }
};
```

Using a Template

```
Stack<char> cs; // Creates code specialized for char
cs.push('a');
char c = cs.pop();

Stack<double*> dps; // Creates version for double*
double d;
dps.push(&d);
```

Implementing Inheritance

Simple: Add new fields to end of the object

Fields in base class always at same offset in derived class

Consequence: Derived classes can never remove fields

C++

```
class Shape {
  double x, y;
};

class Box : Shape {
  double h, w;
};
```

Equivalent C

```
struct Shape {
  double x, y;
};

struct Box {
  double x, y;
  double h, w;
};
```

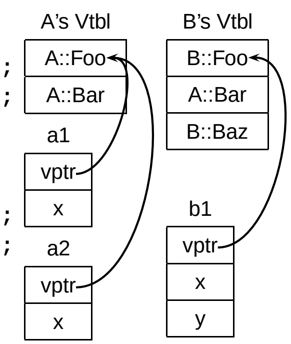
Virtual Functions

```
class Shape {
  virtual void draw(); // Invoked by object's class
};                     // not its compile-time type.
class Line : public Shape {
  void draw();
};
class Arc : public Shape {
  void draw();
};

Shape *s[10];
s[0] = new Line;
s[1] = new Arc;
s[0]->draw();    // Invoke Line::draw()
s[1]->draw();    // Invoke Arc::draw()
```

Virtual Functions

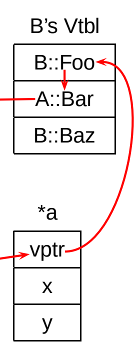
The Trick: Add a “virtual table” pointer to each object.

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};

A a1, a2; B b1;
```



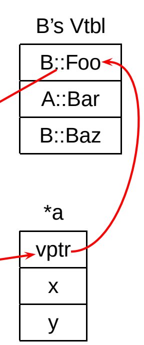
Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar()
    { do_something(); }
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};
A *a = new B;
a->Bar();
```



Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo()
    { something_else(); }
  virtual void Baz();
};
A *a = new B;
a->Foo();
```



Multiple Inheritance

Rocket Science,
and nearly as dangerous

Inherit from two or more classes

```
class Window { ... };

class Border { ... };

class BWindow : public Window,
                public Border {
  ⋮
};
```

Multiple Inheritance Ambiguities

```
class Window {
  void draw();
};

class Border {
  void draw();     // OK
};

class BWindow : public Window,
                public Border { };

BWindow bw;
bw.draw();     // Compile-time error: ambiguous
```

Resolving Ambiguities Explicitly

```
class Window { void draw(); };

class Border { void draw(); };

class BWindow : public Window,
                public Border {
  void draw() { Window::draw(); }
};

BWindow bw;
bw.draw();     // OK
```

Duplicate Base Classes

A class may be inherited more than once

```
class Drawable { ... };
class Window : public Drawable { ... };
class Border : public Drawable { ... };
class BWindow : public Window, public
Border { ... };
```

BWindow gets two copies of the Drawable base class.

Virtual Base Classes

Virtual base classes are inherited at most once

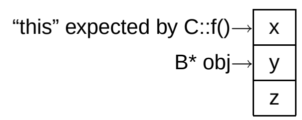
```
class Drawable { ... };
class Window : public virtual Drawable {
... };
class Border : public virtual Drawable {
... };
class BWindow : public Window, public
Border { ... };
```

BWindow gets two copies of the Drawable base class

Implementing Multiple Inheritance

A virtual function expects a pointer to its object

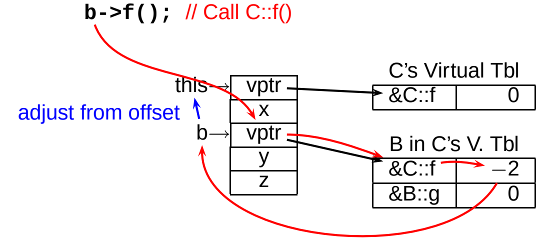
```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f(); }
struct C : A, B { int z; void f(); }

B *obj = new C;
b->f(); // Calls C::f()
```



“obj” is, by definition, a pointer to a B, not a C. Pointer must be adjusted depending on the actual type of the object. At least two ways to do this.

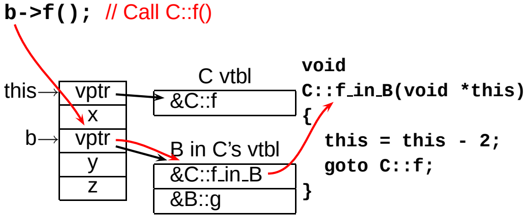
Implementation using Offsets

```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
                  virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()
```



Implementation using Thunks

```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
                  virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()
```



Offsets vs. Thunks

Offsets	Thunks
Offsets to virtual tables	Helper functions
Can be implemented in C	Needs “extra” semantics
All virtual functions cost more	Only multiply-inherited functions cost
Tricky	Very Tricky

Exceptions

A high-level replacement for C's setjmp/longjmp.

```
struct Except { };

void baz() { throw Except; }
void bar() { baz(); }

void foo() {
  try {
    bar();
  } catch (Except e) {
    printf("oops");
  }
}
```

One Way to Implement Exceptions

```
try {                    push(Ex, Handler);

  throw Ex;              throw(Ex);
                         pop();
                         goto Exit;
} catch (Ex e) {       Handler:
  foo();                 foo();
}                      Exit:
```

`push()` adds a handler to a stack

`pop()` removes a handler

`throw()` finds first matching handler

Problem: imposes overhead even with no exceptions

Implementing Exceptions Cleverly

Real question is the nearest handler for a given PC.

```
1  void foo() {
2
3    try {
4      bar();
5    } catch (Ex1 e) { H1: a(); }
6
7  }
8  void bar() {
9
10   try {
11     throw Ex1();
12   } catch (Ex2 e) { H2: b(); }
13
14 }
```

1. look in table

2. H2 doesn't handle Ex1, reraise

3. look in table

Lines	Action
1–2	Reraise
3–5	H1
6–9	Reraise
10–12	H2
13–14	Reraise

Standard Template Library

I/O Facilities

- iostream, fstream

Garbage-collected String class

Containers

- vector, list, queue, stack, map, set

Numerical

- complex, valarray

General algorithms

- search, sort

C++ I/O

C's printing facility is clever but not type safe.

```
char *s; int d; double g;
printf("%s %d %g", s, d, g);
```

Hard for compiler to typecheck argument types against format string.

C++ overloads the **<<** and **>>** operators. This is type safe.

```
cout << 's' << ' ' << d << ' ' << g;
```

C++ I/O

Easily extended to print user-defined types

```
ostream &
operator <<(ostream &o, MyType &m) {
  o << "An Object of MyType";
  return o;
}
```

Input overloads the **>>** operator

```
int read_integer;
cin >> read_integer;
```

C++ String Class

Provides automatic garbage collection, usually by reference counting.

```
string s1, s2;
s1 = "Hello";
s2 = "There";
s1 += " goodbye";
s1 = ""; // Frees memory holding "Hello goodbye"
```

C++ STL Containers

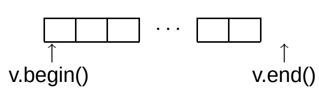
Vector: dynamically growing and shrinking array of elements.

```
vector<int> v;
v.push_back(3); // vector can behave as a stack
v.push_back(2);
int j = v[0];    // operator[] defined for vector
```

Iterators

Mechanism for stepping through containers

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



Associative Containers

Keys must be totally ordered

Implemented with trees—O(log n)

Set of objects

```
set<int, less<int> > s;
s.insert(5);
set<int, less<int> >::iterator i =
s.find(3);
```

Map: Associative array

```
map<int, char*> m;
m[3] = "example";
```

C++ In Embedded Systems

Dangers of using C++:

No or bad compiler for your particular processor

Increased code size

Slower program execution

Much harder language to compile

Unoptimized C++ code can be larger & slower than equivalent C

C++ Features With No Impact

Classes

- Fancy way to describe functions and structs
- Equivalent to writing object-oriented C code

Single inheritance

- More compact way to write larger structures

Function name overloading

- Completely resolved at compile time

Namespaces

- Completely resolved at compile time

Inexpensive C++ Features

Default arguments

- Compiler adds code at call site to set default arguments
- Long argument lists costly in C and C++ anyway

Constructors and destructors

- Function call overhead when an object comes into scope (normal case)
- Extra code inserted when object comes into scope (inlined case)

Medium-cost Features

Virtual functions

- Extra level of indirection for each virtual function call
- Each object contains an extra pointer

References

- Often implemented with pointers
- Extra level of indirection in accessing data
- Can disappear with inline functions

Inline functions

- Can greatly increase code size for large functions
- Usually speeds execution

High-cost Features

Multiple inheritance

- Makes objects much larger (multiple virtual pointers)
- Virtual tables larger, more complicated
- Calling virtual functions even slower

Templates

- Compiler generates separate code for each copy
- Can greatly increase code sizes
- No performance penalty

High-cost Features

Exceptions

- Typical implementation:
- When exception is thrown, look up stack until handler is found and destroy automatic objects on the way
- Mere presence of exceptions does not slow program
- Often requires extra tables or code to direct clean-up
- Throwing and exception often very slow

High-cost Features

Much of the standard template library

- Uses templates: often generates lots of code
- Very dynamic data structures have high memory-management overhead
- Easy to inadvertently copy large data structures

The bottom line

C still generates better code

Easy to generate larger C++ executables

Harder to generate slower C++ executables

Exceptions most worrisome feature

- Consumes space without you asking
- GCC compiler has a flag to enable/disable exception support **`-fexceptions`** and **`-fno-exceptions`**