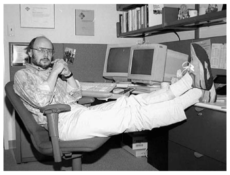
The C++ Language

Prof. Stephen A. Edwards

The C++ Language

- **Bjarne Stroupstrup, the language's creator**

  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**

Example: A stack in C

```
typedef struct {
  char s[SIZE];
  int  sp;
} Stack;


stack *create() {
  Stack *s;
  s = (Stack *)malloc(sizeof(Stack));
  s->sp = 0;
  return s;
}
```

Creator function ensures stack is created properly.

Does not help for stack that is automatic variable.

Programmer could inadvertently create uninitialized stack.

Example: A stack in C

```
char pop(Stack *s) {
  if (sp = 0) error(“Underflow”);
  return s->s[--sp];
}

void push(Stack *s, char v) {
  if (sp == SIZE) error(“Overflow”);
  s->s[sp++] = v;
}
```

Not clear these are the only stack-related functions.

Another part of program can modify any stack any way it wants to, destroying invariants.

Temptation to inline these computations, not use functions.

C++ Solution: Class

```
class Stack {
  char s[SIZE];
  int sp;

public:

  Stack() { sp = 0; }
  void push(char v) {
    if (sp == SIZE) error(“overflow”);
    s[sp++] = v;
  }
  char pop() {
    if (sp == 0) error(“underflow”);
    return s[--sp];
  }
};
```

Definition of both representation and operations

Public: visible outside the class

Constructor: initializes

Member functions see object fields like local variables

C++ Stack Class

- **Natural to use**

```
Stack st;
st.push('a'); st.push('b');
char d = st.pop();

Stack *stk = new Stack;
stk->push('a'); stk->push('b');
char d = stk->pop();
```

C++ Stack Class

- **Members (functions, data) can be public, protected, or private**

```
class Stack {
  char s[SIZE];
public:
  char pop();
};

Stack st;
st.s[0] = 'a';     // Error: sp is private
st.pop();          // OK
```

Class Implementation

- **C++ compiler translates to C-style implementation**

C++ **Equivalent C implementation**

```
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

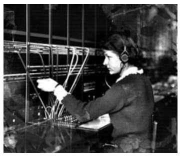
- **For manipulating user-defined "numeric" types**

```
Complex c1(1,5.3), c2(5);

Complex c3 = c1 + c2;
c3 = c3 + 2.3;
```

Creating objects of the user-defined type

Want + to mean something different in this context

Promote 2.3 to a complex number here



Example: Complex number type

- **C++'s operator overloading makes it elegant**

```
Class Complex {
  double re, im;
public:
  complex(double);
  complex(double, double);

  complex& operator+=(const complex&);
};
```

Pass-by-reference reduces copying

Operator overloading defines arithmetic operators for the complex type

References

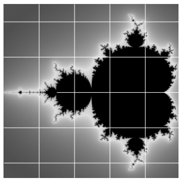
- **Designed to avoid copying in overloaded operators**
- **A mechanism for calling functions pass-by-reference**
- **C only has pass-by-value**

```
void swap(int x, int y) {        /* Doesn't work */
  int tmp = x; x = y; y = tmp;
}
void swap(int &x, int &y) {    /* Works with references */
  int tmp = x; x = y; y = tmp;
}
```

Complex Number Type

- **Member functions including operators can be defined inside or outside the class definition**

```
Complex&
Complex::operator+=(const complex &a)
{
  re += a.re;
  im += a.im;
  return *this;
}
```



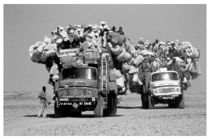
Complex Number Class

- **Operators can also be defined outside classes**

```
Complex operator+(const Complex a,
                  const Complex b) {
  Complex sum = a;  // Copy constructor
  a += b;           // invoke Complex::operator +=
  return sum;
}
```

Function Overloading

- **Overloaded operators a specific case of overloading**
- **General: select specific method/operator based on name, number, and type of arguments**



```
void foo(int);
void foo(int, int);   // OK
void foo(char *);     // OK
int foo(char *);      // BAD: return type not in signature
```

Const

Rock of Gibraltar

- **Access control over variables, arguments.**
- **Provides safety**

```
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

- **Our stack type is nice, but hard-wired for a single type of object**
- **Using array of “void *” or a union might help, but breaks type safety**
- **C++ solution: a template class**
- **Macro-processor-like way of specializing a class to specific types**
- **Mostly intended for container classes**
- **Standard Template Library has templates for**
  - **strings, lists, vectors, hash tables, trees, etc.**

Template Stack Class

```
template <class T> class Stack {
  T s[SIZE];
  int sp;

public:

  Stack() { sp = 0; }
  void push(T v) {
    if (sp == SIZE) error(“overflow”);
    s[sp++] = v;
  }
  T pop() {
    if (sp == 0) error(“underflow”);
    return s[--sp];
  }
};
```

T is a type argument

Used like a type within the body

Using a template

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

Stack<double *> dps;
double d;
dps.push(&d);
```

Display-list example

- Say you want to draw a graphical scene
- List of objects
  - lines, arcs, circles, squares, etc.
- How do you store them all in a single array?

```
void *list[10];                 // Ugly: type-unsafe
```

- How do you draw them all?

```
switch (object->type) {         // Hard to add new object
case LINE: /* … */ break;
case ARC: /* … */ break;
}
```

Inheritance

- Inheritance lets you build derived classes from base classes

```
class Shape { /* … */ };
class Line : public Shape { /* … */ };    // Also a Shape
class Arc : public Shape { /* … */ };     // Also a Shape

Shape *dlist[10];
```

Inheritance

```
class Shape {
  double x, y;          // Base coordinates of shape
public:
  void translate(double dx, double dy) {
     x += dx; y += dy;
  }
};

class Line : public Shape {
};

Line l;
l.translate(1,3);               // Invoke Shape::translate()
```

Line inherits both the representation and member functions of the Shape class

Implementing Inheritance

- Add new fields to the end of the object
- Fields in base class at same offset in derived class

C++

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

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

Equivalent C implementation

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

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

Virtual Functions

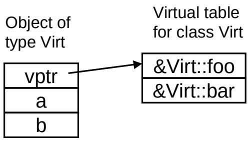
```
class Shape {
  virtual void draw();
};
class Line : public Shape {
  void draw();
};
class Arc : public Shape {
  void draw();
};

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

draw() is a virtual function invoked based on the actual type of the object, not the type of the pointer

New classes can be added without having to change "draw everything" code

Implementing Virtual Functions

- Involves some overhead



```
class Virt {
  int a, b;
  virtual void foo();
  virtual void bar();
};
```

C++

```
void f(Virt *v)
{
  v->bar();
}
```

Equivalent C implementation

```
void f(Virt *v)
{
  (*(v->vptr.bar))(v);
}
```

Cfront

- How the language was first compiled
- Full compiler that produced C as output
- C++ semantics therefore expressible in C

- C++ model of computation ultimately the same
- C++ syntax substantial extension of C
- C++ semantics refer to the same model as C

- So why use C++?
  - Specifications are clearer, easier to write and maintain

Default arguments

- Another way to simplify function calls
- Especially useful for constructors

```
void foo(int a, int b = 3, int c = 4) { /* … */ }
```

C++	Expands to
foo(3)	foo(3,3,4)
foo(4,5)	foo(4,5,4)
foo(4,5,6)	foo(4,5,6)

Declarations may appear anywhere

- Convenient way to avoid uninitialized variables

```
void f(int i, const char *p)
{
  if (i<=0) error();
  const int len = strlen(p);
  char c = 0;
  for (int j = i ; j<len ; j++)
    c += p[j];
}
```

Multiple Inheritance

- Rocket Science

- Inherit from two or more classes:

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

Multiple Inheritance Ambiguities

- What happens with duplicate methods?

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

BWindow bw;
bw.draw();          // Error: ambiguous
```

Multiple Inheritance Ambiguities

- **Ambiguity can be resolved explicitly**

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

BWindow bw;
bw.draw();    // BWindow::draw() calls Window::draw()
```

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**

Duplicate 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 { … };
```

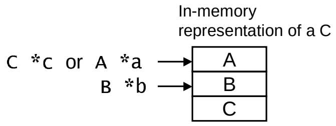
- **BWindow gets one copy of the Drawable base class**

Implementing Multiple Inheritance

- **A virtual function expects a pointer to its object**

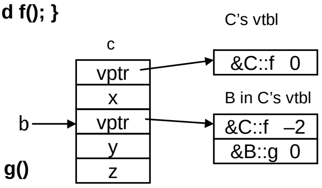
```
struct A { virtual void f(); }
struct B { virtual void f(); }
struct C : A, B { void f(); }
```

- **E.g., C::f() expects "this" to be a C***
- **But this could be called with "this" being a B***



Implementation Using VT Offsets

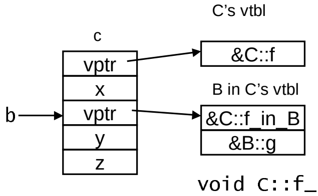
```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f(); virtual void g(); }
struct C : A, B { int z; void f(); }
C c;
B *b = &c;
b->f(); // C::f()
```



1. **b is a B*: vptr has f(), g()**
2. **Call C::f(this – 2)**
3. **First argument now points to a C**

Implementation Using Thunks

- **Create little "helper functions" that adjust** `this`
- **Advantage: Only pay extra cost for virtual functions with multiple inheritance**



```
void C::f_in_B(void* this) {
  return C::f(this – 2);
}
```

Namespaces

- **Namespace pollution**
  - **Occurs when building large systems from pieces**
  - **Identical globally-visible names clash**
  - **How many programs have a “print” function?**
  - **Very difficult to fix**
- **Classes suggest a solution**

```
class A { void f(); };

class B { void f(); };
```

- **Two f’s are separate**

Namespaces

- **Scope for enclosing otherwise global declarations**

```
namespace Mine {
  void print(int);
  const float pi = 3.1415925635;
  class Shape { };
}

void bar(float y) {
  float x = y + Mine::pi;
  Mine::print(5);
}
```

Namespaces

- `using` **directive brings namespaces or objects into scope**

```
namespace Mine {
  const float pi = 3.1415926535;
  void print(int);
}

using Mine::print;
void foo() { print(5); } // invoke Mine::print

using namespace Mine;
float twopi = 2*pi;       // Mine::pi
```

Namespaces

- **Namespaces are open: declarations can be added**

```
namespace Mine {
  void f(int);
}

namespace Mine {
  void g(int);        // Add Mine::g() to Mine
}
```

Namespaces

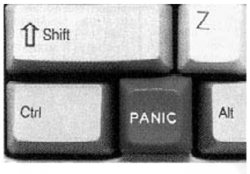
- **Declarations and definitions can be separated**

```
namespace Mine {
  void f(int);
}

void Mine::f(int a) {
 /* … */
}
```

Exceptions

- **A high-level replacement for C’s setjmp/longjmp**

```
struct Except {};

void bar() { throw Except; }

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



Standard Template Library

- I/O Facilities: iostream
- Garbage-collected String class
- Containers
  - vector, list, queue, stack, map, set
- Numerical
  - complex, valarray
- General algorithms
  - search, sort

C++ IO Facilities

- C’s printing facility is clever but unsafe

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

- Hard for compiler to typecheck argument types against format string

- C++ IO overloads the << and >> operators

```
cout << s << ‘ ‘ << d << ‘ ‘ << g;
```

- Type safe

C++ IO Facilities

- Printing 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

- Reference-counted for automatic garbage collection

```
string s1, s2;

s1 = “Hello”;
s2 = “There”;
s1 += “ goodbye”;
s1 = “”; // Frees memory occupied by “Hello goodbye”
```

C++ STL Containers

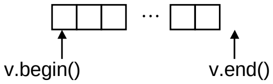
- Vector
  - Dynamically growing, shrinking array of elements

```
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;
}
```



Other Containers

	Insert/Delete from front	 mid.	 end	 random access
vector	O(n)	O(n)	O(1)	O(1)
list	O(1)	O(1)	O(1)	O(n)
deque	O(1)	O(n)	O(1)	O(n)

Associative Containers

- Keys must be totally ordered
- Implemented with trees
- set
  - 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 often much 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 datastructures

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