Introduction to C++
William Knottenbelt (after Nat Pryce)

- Programming in C++ vs. C and Java…
- Programming language features
  - Classes, constructors/destructors, virtual functions
  - Single and multiple inheritance
  - Operator overloading
  - Templates and the standard library
  - Runtime type identification and cast operators
  - Exceptions
- Programming style and idioms

You should know…

- Java programming and O.O. design
  - Basic language features
  - Exceptions
  - inheritance / interfaces / abstract classes
  - Design Patterns
- C Programming
  - Basic language features.
    - Control flow, variables, functions, defining and using types, pointers, malloc and free, casts.

The C++ Language

- Based on C
  - BUT programming style is completely different
- Stricter than C, less strict than Java
- Large and complicated language
  - Many warts and subtle “gotchas”
  - Standards are now mature, but language feature implementation still varies across compilers.

“C lets you shoot yourself in the foot. C++ makes it harder, but when you do, you lose your whole leg”

C++ Classes

```
class Example
{
public:
   // public interface

protected:
   // protected interface

private:
   // private details
};
```

Language Feature

A C++ class is a struct whose members are private by default.

Unlike C, a C++ class or struct can contain both member variables and member *functions*.

A member function is like a Java method: it gets an implicit “this” variable, whose type is a pointer to an object of the class.

C++ Classes

```
class Example
{
public:
   // public interface

protected:
   // protected interface

private:
   // private details
};
```

Language Feature

Unlike Java, visibility qualifiers are specified for blocks, not individual members.

Public members can be accessed by any other class or function.

Protected members can be accessed by functions of derived classes or friends.

Private members can only be accessed by functions of the same class, or friends.

We'll discuss friends later...

C++ Classes

```
class Example
{
public:
   // public interface

protected:
   // protected interface

private:
   // private details
};
```

Language Feature

Class declarations are terminated with a semicolon.

Like a C struct declaration, a class declaration may include in-line variable declarations, but this is much less common in C++ than in C.

C++ Classes

```
class Example
{
public:
   // public interface

protected:
   // protected interface

private:
   // private details
};
```

Style Tip

Unlike Java, a C++ class is usually defined in a *header file* and its methods implemented in a separate source file. Programmers read the header file for reference.

Therefore: define the *public* interface of a class first, because it will be read most often, followed by the *protected* interface, followed by the *private* implementation details, which should not affect users of the class.

Constructors and Destructors

```
class Example
{
public:
   Example();
   ~Example();

private:
   Thing *thing;
};

Example::Example() {
   thing = new Thing;
}

Example::~Example() {
   delete thing;
}
```

Language Feature

A constructor is a special member function that initialises the state of an object.

A destructor releases resources owned by the object when it is destroyed.

This is especially useful for dynamically allocated memory.

C++ does not have garbage collection.

Constructors and Destructors

```
class Example
{
public:
   Example();
   ~Example();

private:
   Thing *thing;
};

Example::Example() {
   thing = new Thing;
}

Example::~Example() {
   delete thing;
}
```

Language Feature

Unlike C, C++ has keywords for allocating ("new") and freeing ("delete") memory.

The delete operator will call the destructor of the object being deleted.

There is a separate operator ("delete[]") for deleting arrays of objects. It calls the destructor of all objects in the array.

Constructors and Destructors

```
class Example
{
public:
   Example();
   ~Example();

private:
   Thing *thing;
};

Example::Example() {
   thing = new Thing;
}

Example::~Example() {
   delete thing;
}
```

C++ Idiom

A *private* constructor or destructor is used to prevent the class being instantiated.

e.g. a *Singleton* class must instantiate and prevent other copies of itself being instantiated.

A *protected* constructor or destructor prevents the class being instantiated but allows it to be inherited.

Copying and Assignment

```
class X
{
public:
   X();
   X( const X& );

   const X &operator= (const X& );

   ~Example();
private:
   // private state
};

X an_x;
X *a_pointer_to_an_x;
```

Language Feature

C++ variables can hold objects *or* pointers to objects.

(Java only allows variables to hold pointers to objects.)

Objects can be passed to a function *by value*, and overwritten by assignment. The semantics of these operations are provided by the *copy constructor* and the *assignment operator*.

We'll discuss operator overloading in detail later…

Copying and Assignment

```
class X
{
public:
   X();
   X( const X& );

   const X &operator= (const X& );

   ~Example();
private:
   // private state
};

X an_x;
X *a_pointer_to_an_x;
```

Style Tip

If a class has a constructor, it should also have a destructor.

If it has a copy constructor, it should also have an assignment operator.

Copy constructors and assignment operators should take a *const* reference to the copied object.

Assignment operators should return a const reference to the assigned object to match the semantics of assignment to primitive types.

Copying and Assignment

```
class X
{
public:
   X();
   X( const X& );

   const X &operator= (const X& );

   ~Example();
private:
   // private state
};

X an_x;
X *a_pointer_to_an_x;
```

Language Feature

A reference in Java is a pointer to an object.

A reference in C++ is an *alias* for an object:

- It is not the same as a pointer.
- You can't take its address or size.
- It might not exist at run-time.
- (It's weird but necessary)

Copying and Assignment

```
class X
{
public:
   X();
   X( const X& );

   const X &operator= (const X& );

   ~Example();
private:
   // private state
};

X an_x;
X *a_pointer_to_an_x;
```

Language Feature

The compiler will automatically generate a copy constructor and assignment operator if you do not define them yourself.

The generated functions copy all members by using their copy constructors or assignment operators. This will probably cause an error if the class uses pointers.

To avoid subtle errors, declare the copy constructor and assignment operator to be private if they do not make sense for a class.

Friends

```
class X {
   friend class Y;
   friend void Func( X & );
public:
   ...
private:
   int stuff;
};


void Func( X &x ) {
   x.stuff = 1;
}
```

Language Feature

"Friends of a class can play with its private parts."

A friend may be a function or a class. All member functions of a friend class may access the private state of the class.

Friendship must be declared within a class declaration. Friendship cannot be inherited.

Therefore friendship ***does not break encapsulation***.

(The closest Java equivalent is package-scope access).

Inheritance & Virtual Functions

```
class Base {
public:
   Base();
   virtual void foo();
   virtual void bar() = 0;
   void baz();
   virtual ~Base() {}
};

class Foo : public Base {
public:
   Foo() : Base() { ... }

   void foo() { Base::foo(); }
   void bar() { ... }
   ~Foo() {}
};
```

Language Feature

C++ classes can inherit interface and behaviour from other classes.

There is no root ancestor class.

Public inheritance exposes public methods of base class at interface of derived class.

Protected inheritance exposes public methods of the base class to derived classes of derived class.

Private inheritance exposes public method of base class to derived class only.

Inheritance & Virtual Functions

```
class Base {
public:
   Base();
   virtual void foo();
   virtual void bar() = 0;
   void baz();
   virtual ~Base() {}
};

class Foo : public Base {
public:
   Foo() : Base() { ... }

   void foo() { Base::foo(); }
   void bar() { ... }
   ~Foo() {}
};
```

Language Feature

Only member functions marked "virtual" are polymorphic.

(This is the opposite to Java, where non-polymorphic methods must be explicitly marked "final".)

Abstract virtual functions are declared by appending "= 0". An abstract class is any class with one or more abstract virtual functions.

A pure abstract class is one in which *all* virtual functions are abstract. This is equivalent to a Java interface.

Inheritance & Virtual Functions

```
class Base {
public:
   Base();
   virtual void foo();
   virtual void bar() = 0;
   void baz();
   virtual ~Base() {}
};

class Foo : public Base {
public:
   Foo() : Base() { ... }

   void foo() { Base::foo(); }
   void bar() { ... }
   ~Foo() {}
};
```

Language Feature

Methods of the base class are invoked using the scope operator ("::").

There is no "super" keyword in C++. It could not work with multiple inheritance.

The constructor of the base class is invoked in the initialisation list of the derived class constructor.

The destructor of the base class is invoked automatically, and should not be explicitly invoked.

Inheritance & Virtual Functions

```
class Base {
public:
   Base();
   virtual void foo();
   virtual void bar() = 0;
   void baz();
   virtual ~Base() {}
};

class Foo : public Base {
public:
   Foo() : Base() { ... }

   void foo() { Base::foo(); }
   void bar() { ... }
   ~Foo() {}
};
```

Style Tip

Like other member functions, destructors are not polymorphic by default.

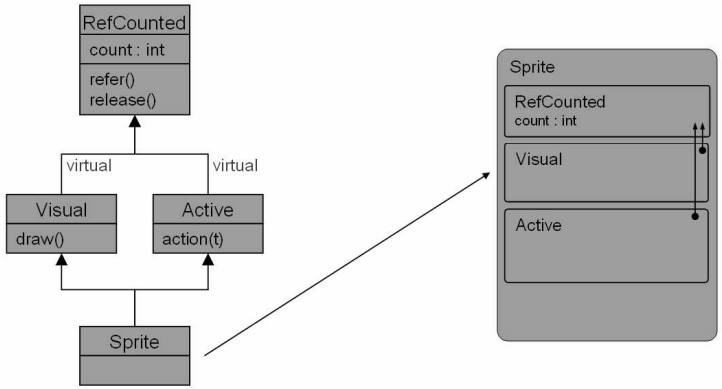
If a polymorphic class does not have a polymorphic destructor, the wrong destructor will be called when the object is deleted through a pointer to its base class.

Therefore: all classes with virtual functions should have a virtual destructor.

Multiple Inheritance

- Classes can inherit from *multiple* bases
- What if the same class is inherited more than once?
  - Usually: multiple copies of inherited state.
  - "Virtual" inheritance: single copy of state.
- Multiple inheritance is (sometimes) fundamental to good C++ design

M.I. Usage: Reference Counting



Type Cast Operators

```
class X; class Y;
X *x;
Y *y;
X &xref;
const X *xconst;
long l;

x = (Y*)y;

x = static_cast<X*>(y);

x = dynamic_cast<Y*>(y);
xref = dynamic_cast<Y&>(*y);

x = const_cast<X*>(xconst);

x = reinterpret_cast<X*>(l);
```

Language Feature

C++ supports the C-style type-cast operator, but its behaviour is dangerous and difficult to understand in the face of multiple and virtual inheritance.

Therefore, C++ provides a number of explicit, and *safe*, type-cast operators.

Type Cast Operators

```
class X; class Y;
X *x;
Y *y;
X &xref;
const X *xconst;
long l;

x = (Y*)y;

x = static_cast<X*>(y);

x = dynamic_cast<Y*>(y);
xref = dynamic_cast<Y&>(*y);

x = const_cast<X*>(xconst);

x = reinterpret_cast<X*>(l);
```

Language Feature

The operator “static_cast<T>(p)” casts value “p” to type “T”.

It does not perform run-time type checking, and so may crash the program if the cast is incorrect.

It cannot cast from a virtual base class to a derived class.

Type Cast Operators

```
class X; class Y;
X *x;
Y *y;
X &xref;
const X *xconst;
long l;

x = (Y*)y;

x = static_cast<X*>(y);

x = dynamic_cast<Y*>(y);
xref = dynamic_cast<Y&>(*y);

x = const_cast<X*>(xconst);

x = reinterpret_cast<X*>(l);
```

Language Feature

The “dynamic_cast” operator performs run-time type checking to ensure that casts between class types are safe.

If applied to a pointer, it returns 0 if the cast is invalid.

If applied to a reference, it throws an exception if the cast is invalid.

The “dynamic_cast” operator *can* cast from a virtual base class to a derived class.

Type Cast Operators

```
class X; class Y;
X *x;
Y *y;
X &xref;
const X *xconst;
long l;

x = (Y*)y;

x = static_cast<X*>(y);

x = dynamic_cast<Y*>(y);
xref = dynamic_cast<Y&>(*y);

x = const_cast<X*>(xconst);

x = reinterpret_cast<X*>(l);
```

Language Feature

The "const_cast" operator casts away const-ness from a value.

This can crash the program if used on global const variable that the compiler has placed in read-only memory.

Type Cast Operators

```
class X; class Y;
X *x;
Y *y;
X &xref;
const X *xconst;
long l;

x = (Y*)y;

x = static_cast<X*>(y);

x = dynamic_cast<Y*>(y);
xref = dynamic_cast<Y&>(*y);

x = const_cast<X*>(xconst);

x = reinterpret_cast<X*>(l);
```

Language Feature

The "reinterpret_cast" operator interprets the bit pattern of a value of one type as a value of another type.

It's behaviour is implementation dependent!

It is mostly used for interpreting data returned from calls to the operating system or received from hardware.

Operator Overloading

- C++ operators can be overloaded for user-defined types
- Mostly for assignment and comparison op's
- Best for math types
- Cannot define new operators or change precedence
- Can easily be misused

Style Tip

Don't define operators to behave differently than their mathematical semantics.

Doing so makes code difficult to understand.

A common exception: stream input and output operators.

Operator Overloading

- C++ operators can be overloaded for user-defined types
- Mostly for assignment and comparison op's
- Best for math types
- Cannot define new operators or change precedence
- Can easily be misused

C++ Idiom

Standard library defines stream input and output operators:

operator << = output

operator >> = input

Can be overloaded for user-defined types.

Acceptable to overload these operators for user-defined I/O abstractions. e.g. binary I/O.

Idiom: Smart Pointers

```
class SpritePtr {
public:
    SpritePtr() : s(0) {}
    SpritePtr( const SpritePtr &p );
    SpritePtr( Sprite *p );

    Sprite *operator->() { return s; }
    Sprite &operator*() { return *s; }

    const SpritePtr&
    operator = (const SpritePtr &p );

    ~SpritePtr();
private:
    Sprite *s;
};

bool operator == ( const SpritePtr &p1,
                   const SpritePtr &p2 );
```

C++ Idiom

Any class that implements the dereference operators ("->" and "*") can be used as a pointer.

The "->" operator should return a pointer or an object with a "->" operator.

Such classes are termed "smart" pointers, because they can do more than a primitive pointer:

- Delete object when destroyed
- Reference-counted G.C
- Lazy instantiation

Idiom: Smart Pointers

```
class SpritePtr {
public:
    SpritePtr() : s(0) {}
    SpritePtr( const SpritePtr &p );
    SpritePtr( Sprite *p );

    Sprite *operator->() { return s; }
    Sprite &operator*() { return *s; }

    const SpritePtr&
    operator = (const SpritePtr &p );

    ~SpritePtr();
private:
    Sprite *s;
};

bool operator == ( const SpritePtr &p1,
                   const SpritePtr &p2 );
```

```
SpritePtr::SpritePtr
    ( const SpritePtr &p ):
    s(p.s) {
    s->refer();
}

const SpritePtr&
SpritePtr::operator =
    (const SpritePtr &p )
{
    p.s->refer();
    s->release();
    s = p.s;
}

SpritePtr::~SpritePtr() {
    s->release();
}
```

Idiom: Smart Pointers

```
class SpritePtr {
public:
    SpritePtr() : s(0) {}
    SpritePtr( const SpritePtr &p );
    SpritePtr( Sprite *p );

    Sprite *operator->() { return s; }
    Sprite &operator*() { return *s; }

    const SpritePtr&
    operator = (const SpritePtr &p );

    ~SpritePtr();
private:
    Sprite *s;
};

bool operator == ( const SpritePtr &p1,
                   const SpritePtr &p2 );
```

Style Tip

A smart pointer should implement both the "->" and "*" operators.

If "operator ->" returns type X*, "operator *" should return type X&.

Idiom: Smart Pointers

```
class SpritePtr {
public:
    SpritePtr() : s(0) {}
    SpritePtr( const SpritePtr &p );
    SpritePtr( Sprite *p );

    Sprite *operator->() { return s; }
    Sprite &operator*() { return *s; }

    const SpritePtr&
    operator = (const SpritePtr &p );

    ~SpritePtr();
private:
    Sprite *s;
};

bool operator == ( const SpritePtr &p1,
                   const SpritePtr &p2 );
```

Style Tip

Comparison operators should be declared outside of the class. This provides greater flexibility.

e.g. one can then pass a pointer to a comparison operator to a sort function.

Templates

```
template <typename T>
void exchange(T &a, T &b)
{
  T c;
  c = a;
  a = b;
  b = c;
}

int main() {
  int a = 7, b = 5;
  exchange<int>(a,b);
  char p = 'p', q = 'q';
  exchange<char>(p,q);

  cout << "a=" << a << " b=" << b << endl;
  cout << "p=" << p << " q=" << q << endl;

  return 0;
}
```

Language Feature

A "template" class or function is parameterised at compile-time.

Template parameters can be type names or a compile-time constants of primitive types.

Templates are useful for writing type-safe container classes (like Java generics)...

... they can be complicated...

...but there are many cunning things you can do with templates.

Standard Library

- The standard library makes heavy use of templates
- It provides:
  - Basic types, string classes, platform limits etc.
  - Stream I/O templates
  - Templates for containers and algorithms
  - Functions from the C standard library.

C++ programs usually use containers and strings rather than raw arrays and pointers.

Standard Template Library

```
void print( int n ) {
    cout << n << endl;
}

vector<int> vi = ...


sort( vi.begin(), vi.end() );


vector<int>::iterator i = vi.begin();
while( i != vi.end() ) {
    print(i);
    i++;
}


for_each( vi.begin(), vi.end(), print );
```

Information

Template vector<T> is a one-dimensional array of objects.

Information

Sort is a template function. It can sort a sequence of objects in any container.

The sequence is defined by two iterators.

Standard Template Library

```
void print( int n ) {
    cout << n << endl;
}

vector<int> vi = ...


sort( vi.begin(), vi.end() );


vector<int>::iterator i = vi.begin();
while( i != vi.end() ) {
    print(i);
    i++;
}


for_each( vi.begin(), vi.end(), print );
```

C++ Idiom

STL iterators act like pointers.

STL algorithm templates can be used with both containers and pointers into low-level arrays.

Standard Template Library

```
void print( int n ) {
    cout << n << endl;
}

vector<int> vi = ...


sort( vi.begin(), vi.end() );


vector<int>::iterator i = vi.begin();
while( i != vi.end() ) {
    print(i);
    i++;
}


for_each( vi.begin(), vi.end(), print );
```

Information

STL allows a functional style of programming, including the use of higher-order programming and curried functions.

However, compared to a real functional programming language, the syntax is hideous!

Idiom: Expression Templates

```
template <unsigned int N>
class Bit {
public:
    enum Mask { mask = (1<<N) };
};


enum Flags {
    READ   = Bit<0>::mask,
    WRITE  = Bit<1>::mask,
    APPEND = Bit<2>::mask,
    CREATE = Bit<3>::mask
};
```

C++ Idiom

Template classes can be used to calculate numeric expressions at compile-time.

This is a very simple example.

Recursive “expression templates” can unroll matrix multiplications, such as fast fourier transforms, to massively improve performance

Pointers to Member Functions

```
struct X {
    void foo();
    int *bar( int *bar );
};

X x;
int *ip = 0, *jp = new int(2);

// Declare pointers to member functions
void (X::*mem_fun_1)();
int *(X::*mem_fun_2)( int* );

mem_fun_1 = &X::foo;
mem_fun_2 = &X::bar;

(x.*mem_fun_1)();
ip = (x.*mem_fun_2)( jp );
```

Language Feature

In C, one can take the address of variables and functions.

C++ also allows one to take the address of member variables and member functions of a class.

You need an instance of a class in order to dereference pointers to its members.

Pointers to virtual member functions are themselves polymorphic.

The type declarations are usually typedef’d to a readable name

Exceptions

- C++ uses exceptions to report errors.
- Unlike Java:
  - Functions do not have to declare the exceptions that can be thrown.
  - no garbage collection and no “finally” blocks.
- How to avoid memory leaks?
  - Pointers returned by new must be managed by a smart pointer.