Developed as an *extension* to C
by adding object oriented constructs originally found in Smalltalk (and Simula67).

Most legal C programs are also legal C++ programs

- “Backwards compatibility” made it easier for C++ to be accepted by the programming community
- ... but made certain features problematic (leading to “dirty” programs)

Many of C++ features have been used in Java

- Some have been “cleaned up”
- Some useful features have been left out
C++ and Java: The Commonalities

- Classes, instances (objects), data members (fields) and member functions (methods).
- Overloading and inheritance.
  - base class (C++) → superclass (Java)
  - derived class (C++) → subclass (Java)
- Constructors
- Protection (visibility): private, protected and public
- Static binding for data members (fields)

A C++ Primer for Java Programmers

Classes, fields and methods:

<table>
<thead>
<tr>
<th>Java:</th>
<th>C++:</th>
</tr>
</thead>
<tbody>
<tr>
<td>class A extends B {</td>
<td>class A : public B {</td>
</tr>
<tr>
<td>private int x;</td>
<td>private: int x;</td>
</tr>
<tr>
<td>protected int y;</td>
<td>protected: int y;</td>
</tr>
<tr>
<td>public int f() {</td>
<td>public: int f() {</td>
</tr>
<tr>
<td>return x;</td>
<td>return x;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public void print() {</td>
<td>void print() {</td>
</tr>
<tr>
<td>System.out.println(x);</td>
<td>std::cout &lt;&lt; x &lt;&lt; std::endl;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

A C++ Primer for Java Programmers

Declaring objects:
- In Java, the declaration A va declares va to be a reference to object of class A.
  - Object creation is always via the new operator
- In C++, the declaration A va declares va to be an object of class A.
  - Object creation may be automatic (using declarations) or via new operator:
    A *va = new A;
Objects and References

- In Java, all objects are allocated on the heap; references to objects may be stored in local variables.
- In C++, objects are treated analogous to C structs: they may be allocated and stored in local variables, or may be dynamically allocated.
- Parameters to methods:
  - Java distinguishes between two sets of values: primitives (e.g. ints, floats, etc.) and objects (e.g. String, Vector, etc.).
    - Primitive parameters are passed to methods by value (copying the value of the argument to the formal parameter)
    - Objects are passed by reference (copying only the reference, not the object itself).
  - C++ passes all parameters by value unless specially noted.

Inheritance, Overloading, and Overriding

- **Inheritance**: Subclass inherits all data members and member functions (and can access all public/protected members) from its superclass.
  - Code reuse: If a method f() is defined in class A, and B is a subclass of A ...
    ... the method can be applied to objects of type B without redefinition.
- **Overloading**: A method is distinguished by its name and its signature (the number and types of arguments).
  - So multiple methods can be defined with the same name.
- **Overriding**: A member (field or method) can be redefined in a subclass which will then override access to the same member of the superclass.

Overloading

- Consider the following definition of Java class Test
  ```java
  class Test extends Base {
      void h(Test t);
      void h(Base b);
  }
  ```
- Let t and b refer to objects of class “Base” and “Test” respectively.
- What is the behavior of the following calls?
  ```java
  t.h(b);
  t.h(t);
  ```
Inheritance

Consider the following Java class definitions:

class Base {
    void h(Base b);
}
class Test extends Base {
    void h(Base b);
}

Let \( b \) and \( t \) refer to objects of class \( \text{Base} \) and \( \text{Test} \) respectively.

What is the behavior of the following calls?

\[ b.h(b); \]
\[ t.h(b); \]

Inheritance and Overloading

Instance methods in OO languages have an \textit{implicit} object parameter (i.e. \texttt{this}).

\textit{Inheritance resembles overloading on the implicit parameter.}

Main point to consider:

- What types are used to resolve the overloading?
  
  (i.e., How is the signature of the call constructed?)

Let \( \text{Test} \) be a subclass of \( \text{Base} \). Consider the following definitions:

\[
\text{Base } b; \\
\text{Test } t;
\]

- What are the types of variables \( b \) and \( t \)?
- What are the types of objects that can be referenced by \( b \) and \( t \)?

Types

\textbf{Apparent Type}: Type of an object as per the declaration in the program.

\textbf{Actual Type}: Type of the object at run time.

Let \( \text{Test} \) be a subclass of \( \text{Base} \). Consider the following program:

\[
\begin{align*}
\text{Base } b &= \text{new } \text{Base}(); \\
\text{Test } t &= \text{new } \text{Test}(); \\
\ldots \\
\text{b} &= \text{t};
\end{align*}
\]


\begin{center}
\begin{tabular}{|c|c|}
\hline
Variable & Apparent type of object referenced \\
\hline
\texttt{b} & \text{Base} \\
\texttt{t} & \text{Test} \\
\hline
\end{tabular}
\end{center}

\ldots throughout the scope of \( b \) and \( t \)'s declarations
Let **Test** be a subclass of **Base**. Consider the following program fragment:

```java
Base b = new Base();
Test t = new Test();
...
b = t;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Program point</th>
<th>Actual type of object referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>before b=t</td>
<td>Base</td>
</tr>
<tr>
<td>t</td>
<td>before b=t</td>
<td>Test</td>
</tr>
<tr>
<td>b</td>
<td>after b=t</td>
<td>Test</td>
</tr>
<tr>
<td>t</td>
<td>after b=t</td>
<td>Test</td>
</tr>
</tbody>
</table>

### Binding field and method names

- In Java:
  - field names are resolved using their *apparent* types (i.e., at compile time)  
    [also called “Static Binding”]
  - method names are resolved using their *actual* types (i.e., at run time)  
    [also called “Dynamic Binding”]

- In C++:
  - both field and names are resolved using their *apparent* types (i.e., at compile time)
  - ... *unless methods are declared as virtual and are accessed via references.*

### Polymorphism

*“The ability to assume different forms”*

- A function/method is polymorphic if it can be applied to values of many types.
- Class hierarchy and inheritance provide a form of polymorphism called *subtype polymorphism.*
  [same function can be applied to different types]
- Overloading provides a form of polymorphism called *ad-hoc polymorphism.*
  [different forms are distinguished by types of parameters (sometimes return values too)]
- Polymorphic functions increase code reuse.
Consider the following code fragment: 

\[(x < y) \text{? } x : y\]

“Finds the minimum of two values”.

The same code fragment can be used regardless of whether \(x\) and \(y\) are

- ints
- floats
- (in C++:) in any class that implements operator “<”.

Templates lift the above form of polymorphism (called parametric polymorphism) to functions and classes.

---

**Function Template**

- Declaring function templates:

  ```
  template <typename T>
  T min ( T x, T y ) {
      return (x < y)? x : y;
  }
  ```

  - typename parameter can be name of any type (e.g. int, long, Base, ...)
  - Using template functions:
    - \(z = \text{min}(x, y)\)
    - Compiler fills out the template's typename parameter using the types of arguments.
    - Can also be explicitly used as: \(\text{min<float>}(x, y)\)

---

**Class Templates**

- Of great importance in implementing data structures (say list of elements, where all elements have to be of the same type).

Java does not provide templates:

- Some uses of templates can be replaced by using Java interfaces.
- Many other uses would require “type casting”
  - e.g:
    ```
    Iterator e = ... 
    Int x = (Integer) e.next();
    ```
- Inherently dangerous since it skirts around compile-time type checking.
A **class** declaration (set of (and type of) data members, and signatures of member functions) can be separated into a separate **header** file.

- Header file specifies an "interface".

Member functions and constructors can be defined within a class declaration, or (usually) in separate files (sometimes called *Dot-C files*)

- Dot-C file specifies an "implementation".
- Header files may be included in Dot-C files using the `#include` directive.

Makefiles are used to compile and link program units.