C++ Overview

• What is C++?
• Origination and Evolution of C++
• Why Use C++?
• How Does C++ Differ from Java?
• C++ and Java Minimal Examples
• Compiling C++

What is C++?

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer.


What is C++ (cont’d)?

• Based on C
  – Supports procedural programming paradigm
  – Can link with compiled C code (and libraries)
  – Portable (using preprocessor)
• Adds polymorphism
  – Run-time (dynamic) binding of function calls
• Adds inheritance
  – Reuse interfaces
  – Reuse implementations
What is C++ (cont’d)?

- Adds generic code (template class) support
- Adds exception handling
- Supports large-scale programming
  - Separate compilation
  - Namespaces
  - Libraries (archives)

Origination of C++

- Designed in early 1980’s by Bjarne Stroustrup of Bell Labs
- Backward compatible with C, as much as possible
  - First “compiler”, cfront, actually translated C++ to C
- Improvements over C
  - Stronger typechecking
  - Supports data abstraction
  - Supports object-oriented programming
  - Supports generic programming

Evolution of C++

- Added namespaces, exception handling, run-time type identification (RTTI), improved templates, etc.
- Improved compilers
- Added Standard Template Library (STL) containers and algorithms
- Standardized by ANSI, DIN, BSI, ISO (ISO/IEC 14882)

Why Use C++?

- To maximize execution speed
- To support reuse, with separation of interface and implementation
- To support data abstraction and dynamic binding
- For portability
- For backward source compatibility with C
- For link compatibility with C, Basic, Fortran, Ada, etc.
- To maximize execution speed
### A Stack Example

```cpp
// File Stack.h
typedef int T;
class Stack {
public:
    Stack (size_t size);
    Stack (const Stack &s);
    void operator= (const Stack &s);
    ~Stack (void);
    void push (const T &item);
    void pop (T &item);
    int is_empty (void) const;
    int is_full (void) const;
    // ...
private:
    size_t top_;
    size_t size_;
    T *stack_;}
```

### Manager operations

```cpp
Stack::Stack (size_t size)
    : top_ (0), size_ (size),
    stack_ (new T[size]) {}

Stack::Stack (const Stack &s)
    : top_ (s.top_), size_ (s.size_),
    stack_ (new T[s.size_]) {
    for (size_t i = 0; i < s.size_; i++)
        this->stack_[i] = s.stack_[i];
}

void Stack::operator= (const Stack &s) {
    if (this == &s) return;
    delete [] this->stack_;
    this->stack_ = new T[s.size_];
    this->top_ = s.top_;
    this->size_ = s.size_;
    for (size_t i = 0; i < s.size_; i++)
        this->stack_[i] = s.stack_[i];
}

Stack::~Stack (void) {
    delete [] this->stack_;
}
```

### How Does C++ Differ from Java?

- C++ programs run standalone; the Java interpreter loads and runs any class with a main () method.
- C++ supports generic programming with template classes (definitions) that can separate C++ class interface (header) from implementation.
- C++ allows multiple inheritance of implementations.
- C++ allows memory to be managed by the programmer; it does not provide built-in garbage collection like Java.
- C++ pointer variables access memory.
- C++ passes arguments by value by default.
- C++ arrays are not first class citizens.
- C++ allows operator overloading.
- C++ has a preprocessor; Java relies on the constrained language definition for portability.
- C++ allows global variables, but they should be avoided.
- Built-in C++ types are implementation dependent.
A Stack Example (cont’d)

**Benefits**

1. Data hiding and data abstraction, e.g.,
   ```cpp
   Stack s1 (10), s2 (20), s3 (30);
   ```
2. The ability to declare multiple stack objects
   ```cpp
   Stack s1 (200);
   s1.top_ = 10 // Error flagged by compiler!
   ```
3. Automatic initialization and termination
   ```cpp
   { // Constructor automatically called.
   Stack s1 (1000),
   s2 (1000); // Destructor automatically called.
   } // Destructor automatically called.

**Drawbacks**

1. Error handling is obtrusive. Use exception handling to solve this.
2. The example is limited to a single type of stack element (int in this case).
   ```cpp
   Stack s1 (1), s2 (100);
   T item;
   if (!s1.is_full ())
     s1.push (473);
   if (!s2.is_full ())
     s2.push (2112);
   if (!s2.is_empty ())
     s2.pop (item);
   // Access violation caught // at compile-time!
   s2.top_ = 10;
   // Termination handled automatically // via destructor.
   ```
3. Automatic initialization and termination

---

Accessor and worker operations

```cpp
int Stack::is_empty (void) const {
    return this->top_ == 0;
}
```  

```cpp
int Stack::is_full (void) const {
    return this->top_ == this->size_;}
```  

```cpp
void Stack::push (const T &item) {
    this->stack_[this->top_++] = item;
}
```  

```cpp
void Stack::pop (T &item) {
    item = this->stack_[--this->top_];
}
```
Template Implementation in C++

- A parameterized type Stack class interface using C++

```cpp
// typedef int T;
template <class T>
class Stack {
    public:
        Stack (size_t size);
        ~Stack (void);
        void push (const T &item);
        void pop (T &item);
        // ...
    private:
        size_t top_;
        size_t size_;
        T *stack_;}
```

- A parameterized type Stack class implementation using C++

```cpp
#include "Stack.h"

void foo (void)
{
    Stack<int> s1(1000);
    Stack<float> s2(100);
    s1.push(-291);
    s2.push(3.1416);
}
```

- Another parameterized type Stack class

```cpp
template <class T, size_t SIZE>
class Stack {
    public:
        Stack (void);
        ~Stack (void);
        void push (const T &item);
        void pop (T &item);
        // ...
    private:
        size_t top_;
        size_t size_;
        T stack_[SIZE];
    }
```

- To use:

```cpp
Stack<int, 200> sl;
```
C++ Object-Oriented Features

- C++ provides three characteristics generally associated with object-oriented programming:
  - **Data Abstraction**
    - Package a class abstraction so that only the *public interface* is visible and the *implementation details* are hidden from clients
    - Allow parameterization based on *type*
  - **Single and Multiple Inheritance**
    - A derived class inherits operations and attributes from one or more base classes, possibly providing additional operations and/or attributes

C++ Object-Oriented Features (cont’d)

- **Dynamic Binding**
  - The actual type of an object (and thereby its associated operations) need not be fully known until run-time
  - Compare with C++ *template* feature, which is handled at compile-time
- C++’s object-oriented features encourage designs that
  1. Explicitly distinguish *general properties* of related concepts from
  2. *Specific details* of particular instantiations of these concepts
- *e.g.*, an object-oriented graphical shapes library design using inheritance and dynamic binding
- This approach facilitates extensibility and reusability

Inheritance Preview

- A type can *inherit* or *derive* the characteristics of another *base* type. These derived types act just like the base type, except for an explicit list of:
  1. Operations that are implemented differently, *i.e.*, overridden
  2. Additional operations and extra data members
  3. Modified method access privileges
- C++ supports both single and multiple inheritance, *e.g.:

  ```cpp
  class X { /* . . . */ };  
class Y : public X { /* . . . */ };  
class Z : public X { /* . . . */ };  
class YZ : public Y, public Z { /* . . . */ };  
  ```

Dynamic Binding Preview

- Dynamic binding is a mechanism used along with inheritance to support a form of *polymorphism*
- C++ uses *virtual* functions to implement dynamic binding:
  - The actual method called at run-time depends on the class of the object used when invoking the virtual method
- C++ allows the class definer the choice of whether to make a method virtual or not
  - This leads to time/space performance vs. flexibility tradeoffs
    * Depending on the compiler, virtual methods may introduce a small amount of overhead for each virtual function call
Object-Oriented Implementation in C++
(cont’d)

- Use interface inheritance to create a specialized (i.e., bounded) version of a stack:

```cpp
#include "Stack.h"
#include "Array.h"

template <class T>
class B_Stack : public Stack<T> {
    public:
        B_Stack (size_t size = 100);
        virtual void push (const T &item);
        virtual void pop (T &item);
        virtual int is_empty (void) const;
        virtual int is_full (void) const;
    private:
        Array<T> stack_; // user-defined size_t top_; // built-in
};
```

```
int main (int, char *[]) {
    X x; Y y;
    foo (&x); // X::m, X::vm
    foo (&y); // X::m, Y::vm
}
```

Object-Oriented Implementation in C++
(cont’d)

- class B_Stack implementation

```cpp
template <class T>
B_Stack<T>::B_Stack (size_t size)
    : top_ (0), stack_ (size) {
}

template <class T> void
B_Stack<T>::push (const T &item) {
    this->stack_.set (this->top_++, item);
}

template <class T> void
B_Stack<T>::pop (T &item) {
    this->stack_.get (--this->top_, item);
}

template <class T> int
B_Stack<T>::is_full (void) const {
    return this->top_ >= this->stack_.size ();
}
```

By using "pure virtual methods," we can guarantee that the compiler won’t allow instantiation of a template class for which at least one of the virtual methods is not implemented.

Dynamic Binding Preview (cont’d)

```cpp
class X { // Base class
    public:
        // Non-virtual
        int m (void) {cout << "X::m";} // Virtual
        virtual int vm (void) {cout << "X::vm";}
    }

class Y : public X { // Derived class
    public:
        // Non-virtual
        int m (void) {cout << "Y::m";} // Virtual
        virtual int vm (void) {cout << "Y::vm";}
    }

    void foo (X *x) {
        x->m (); // direct call: _m_1X (x);
        x->vm (); // indirect call: (*x->vptr[1])
    }

    int main (int, char *[]) {
        X x; Y y;
        foo (&x); // X::m, X::vm
        foo (&y); // X::m, Y::vm
    }
```
Likewise, interface inheritance can create a totally different "unbounded" implementation:

```cpp
// Forward declaration.
template <class T> class Node;
template <class T>
class UB_Stack : public Stack<T>
{
public:
  UB_Stack (size_t hint = 100);
  ~UB_Stack (void);
  virtual void push (const T &new_item);
  virtual void pop (T &top_item);
  virtual int is_empty (void) const { return this->head_ == 0; }
  virtual int is_full (void) const { return 0; }
private:
  // Head of linked list of Node<T>'s.
  Node<T> *head_;
};
```

Class UB_Stack implementation:

```cpp
template <class T>
UB_Stack<T>::UB_Stack (size_t hint): head_ (0) {}

template <class T> void
UB_Stack<T>::push (const T &item) {
  Node<T> *t = new Node<T> (item, this->head_);
  assert (t != 0);
  this->head_ = t;
}

template <class T> void
UB_Stack<T>::pop (T &top_item) {
  top_item = this->head_->item_;  
  Node<T> *t = this->head_;  
  this->head_ = this->head_->next_;  
  delete t;
}

template <class T>
UB_Stack<T>::~UB_Stack (void) {
  // delete all Nodes...
  for (T t; this->head_ != 0; this->pop (t)) continue;
}
```

Function and Operator Overloading:

Two or more functions or operators may be given the same name provided the type signatures are unique.

```cpp
void move (int);
void move (int, int);
```

```cpp
doctor /
(const Complex &);
complex /
(const Complex &);
```

A function's return type is not considered when distinguishing between overloaded instances.

```cpp
double operator/ (const Complex &);
complex operator/ (const Complex &);
```
C++ Classes

- The class is the basic data abstraction unit in C++
- The class mechanism facilitates the creation of user-defined Abstract Data Types (ADTs)
  - A class declarator defines a type comprised of data members, as well as method operators
  * Data members may be either built-in or user-defined
  - Classes are "cookie cutters" used to define objects
    * a.k.a. instances

C++ Classes (cont’d)

- For efficiency and C compatibility reasons, C++ has two type systems
  1. One for built-in types, e.g., int, float, char, double, etc.
  2. One for user-defined types, e.g., classes, structs, unions, enums, etc.
- Note that constructors, overloading, inheritance, and dynamic binding only apply to user-defined types
  - This minimizes surprises, but is rather cumbersome to document and explain . . .

C++ Classes (cont’d)

- General characteristics of C++ classes:
  - Any number of class objects may be defined
    * i.e., objects, which have identity, state, and behavior
  - Class objects may be dynamically allocated and deallocated
  - Passing class objects, pointers to class objects, and references to class objects as parameters to functions are legal
  - Vectors of class objects may be defined
- A class serves the same purpose as a Java class, and a similar purpose to a C struct

Class Vector Example

- There are several significant limitations with built-in C and C++ arrays, e.g.,
  - The size must be a compile-time constant, e.g.,
    ```
    void foo (int i) {
        int a[100]; // OK
        int b[i]; // Error!
    }
    ```
  - Array size cannot vary at run-time
  - Legal array bounds run from 0 to size - 1
  - No range checking performed at run-time, e.g.,
    ```
    for (i = 0; i <= 10; i++)
    { ...
    }
    ```
  - Cannot perform full array assignments, e.g.,
    ```
    a = b; // Error!
    ```
Class Vector Interface

// File Vector.h
#include "Vector.h"
#define VECTOR_H

typedef int T;
class Vector {
public:
    Vector (size_t len = 100);
    ~Vector (void);
    size_t size (void) const;
    bool set (size_t i, const T &item);
    bool get (size_t i, T &item) const;

private:
    size_t size_;
    T *buf_;
    bool in_range (size_t i) const;
};
#endif /* VECTOR_H */

Class Vector Implementation

// File Vector.cpp.
#include "Vector.h"

bool Vector::in_range (size_t i) const
{
    return i < this->size ();
}

bool Vector::set (size_t i, const T &item)
{
    if (this->in_range (i)) {
        this->buf_[i] = item;
        return true;
    }
    return false;
}

bool Vector::get (size_t i, T &item) const
{
    if (this->in_range (i)) {
        item = this->buf_[i];
        return true;
    }
    return false;
}
**C++ Objects**

- A C++ object is an instance of a **class** (or any other C++ type for that matter . . .)
- An object can be instantiated or disposed either implicitly or explicitly, depending on its **life-time**
- The life-time of a C++ object is either **static, automatic, or dynamic**
  - C++ refers to this as the **storage class** of an object

**C++ Objects (cont’d)**

- **Life-time or storage class:**
  1. **Static**
     - *i.e.*, it lives throughout life-time of process
     - *static* can be used for local, global, or class-specific objects
       (note, their **scope** is different)
  2. **Automatic**
     - *i.e.*, it lives only during function invocation, on the **run-time stack**
  3. **Dynamic**
     - *i.e.*, it lives between corresponding calls to operators **new** and **delete**
     - Dynamic objects often have life-times that extend beyond the existence of the functions that create them

**C++ Objects (cont’d)**

- Most C++ implementations do **not** support automatic garbage collection of dynamically allocated objects
  - In garbage collection schemes, the **run-time system** is responsible for detecting and deallocating unused dynamic memory
  - Note, it is very difficult to implement garbage collection correctly in C++ due to pointers and unions
- Therefore, programmers **must** explicitly deallocate objects when they want them to go away
  - C++ constructors and destructors are useful for automating certain types of memory management . . .

**Typical layout of memory objects in the process address space**
**C++ Comments**

- C++ allows two commenting styles:
  1. The traditional C bracketed comments, which may extend over any number of lines, e.g., /* This is a multi-line C++ comment */
  2. The "continue until end-of-line" comment style, e.g., // This is a single-line C++ or Java comment
- C-style comments do not nest. However, C++ and C styles nest, so it is possible to comment out code containing other comments, e.g.,

```cpp
/* assert (i < size) // check index range */
// if (i != 0 /* check for zero divide */ && 10 / i)
```

**Const Type Qualifier**

- C++ data objects and methods are qualifiable with the keyword `const`, making them act as *read-only* objects
  - e.g., placing them in the *text segment*
  - `const` only applies to objects, not to types
- Examples
  - `const int max_age = 100;
  - const char question = 'y';`

**Const Type Qualifier (cont’d)**

- User-defined `const` data objects:
  - A `const` qualifier can also be applied to an object of a user-defined type, e.g.,
    ```cpp
    const String string_constant ("Hi, I’m read-only!");
    const Complex complex_zero (0.0, 0.0);
    string_constant = "This will not work!"; // ERROR
    complex_zero += Complex (1.0); // ERROR
    %complex_zero == Complex (0.0); // OK
    ```
  - Ensuring `const correctness` is an important aspect of designing C++ interfaces, e.g.,
    1. It ensures that `const` objects may be passed as parameters
    2. It ensures that data members are not accidentally corrupted

```cpp
const String string_constant ("Read-only");
cout << string_constant.size (); // Fine
string_constant.set (1, 'c'); // Error
```
Stream I/O

- C++ extends standard C library I/O with `stream` and `iostream` classes
- Several goals
  1. **Type-Security**
     - Reduce type errors for I/O on built-in and user-defined types
  2. **Extensibility** (both above and below)
     - Allow user-defined types to interoperate syntactically with existing printing facilities
     - Contrast with printf/scanf-family
     - Transparently add new underlying I/O devices to the iostream model
     - i.e., share higher-level formatting operations

Boolean Type

- C++ has a `bool` built-in type
  - The `bool` values are called `true` and `false`
  - Converting numeric or pointer type to `bool` takes 0 to `false` and anything else to `true`
  - `bool` promotes to `int`, taking `false` to 0 and `true` to 1
  - All operators that conceptually return truth values return `bool`
    - e.g., the operands of
      &

Type Cast Syntax

- C++ introduces a new type cast syntax in addition to Classic C style casts. This function-call syntax resembles the type conversion syntax in Ada and Java, e.g.,
  ```
  // function prototype from math.h library
  double log10 (double param);
  
  /* C style type cast notation */
  if ((int) log10 ((double) 7734) != 0);

  // C++ function-style cast notation
  if (int (log10 (double (7734)))) != 7734);
  
  This “function call” is performed at compile time
  ```

Default Parameters

- C++ allows default argument values in function definitions
  - If trailing arguments are omitted in the actual function call these values are used by default, e.g.,
  ```
  void assign_grade (const char *name,
  const char *grade = 'A');
  ```
  ```
  assign_grade ('Bjarne Stroustrup', 'C++');
  // Bjarne needs to work harder on his tasks
  assign_grade ('Jean Ichbiah');
  // Jean gets an A for Ada!
  ```
  - Default arguments are useful in situations when one must change a class without affecting existing source code
    - e.g., add new params at end of argument list (with default values)
Default Parameters (cont’d)

- Default parameter passing semantics are similar to those in languages like Java:
  - e.g., only trailing arguments may have defaults
    
    // Incorrect
    int x (int a = 10, char b, double c = 10.1);
  - Note, there is no support for named parameter passing

- However, it is not possible to omit arguments in the middle of a call, e.g.,
  int foo (int = 10, double = 2.03, char = 'c');

  foo (100, , 'd'); /* ERROR!!! */
  foo (1000); /* OK, calls foo (1000, 2.03, 'c');

- There are several arcane rules that permit successive redeclarations of a function, each time adding new default arguments

Declaration Statements

- C++ allows variable declarations to occur anywhere statements occur within a block
  - The motivations for this feature are:
    1. To localize temporary and index variables
    2. Ensure proper initialization
  - This feature helps prevent problems like:
    int i, j;
    /* many lines of code . . . */
    // Oops, forgot to initialize!
    while (i < j) /* . . . */;
  - Instead, you can use the following
    for (int i = x, j = y; i < j; )
    /* . . . */;