The C++ Programming Language

A Tour Through C++

Outline

C++ Overview
C++ Design Goals
Major C++ Enhancements
Other Enhancements
Language Features Not Part of C++
Function Prototypes
C++ Classes
Class Vector Example
C++ Objects
C++ Object-Oriented Features

Inheritance Preview

Inheritance Example: Ada-style Vectors

Dynamic Binding Preview

Overloading

New-Style Comments

Type-Safe Linkage

Inline Functions

Dynamic Memory Management

Const Type Qualifier

Stream I/O

Boolean Type

References

Type Cast Syntax

Default Parameters

Declaration Statements

Abbreviated Type Names

User-Defined Conversions

Static Initialization

Miscellaneous Differences

C++ Overview

- C++ was designed at AT&T Bell Labs by Bjarne Stroustrup in the early 80's
  - The original *cfront* translated C++ into C for portability
    * However, this was difficult to debug and potentially inefficient
  - Many native host machine compilers now exist
    * *e.g.*, Borland, DEC, GNU, HP, IBM, Microsoft, Sun, Symantec, etc.

- C++ is a *mostly* upwardly compatible extension of C that provides:
  1. *Stronger typechecking*
  2. *Support for data abstraction*
  3. *Support for object-oriented programming*
     - C++ *supports* the Object-Oriented paradigm but does not *require* it

C++ Design Goals

- As with C, run-time efficiency is important
  - *e.g.*, unlike Ada, complicated run-time libraries have not traditionally been required for C++
    * Note, that there is no language-specific support for concurrency, persistence, or distribution in C++
- Compatibility with C libraries and UNIX tools is emphasized, *e.g.*,
  - Object code reuse
    * The storage layout of structures is compatible with C
    * Support for X-windows, standard ANSI C library, UNIX system calls via **extern** block
  - C++ works with the `make` recompilation utility

C++ Design Goals (cont'd)

- "As close to C as possible, but no closer"
  - *i.e.*, C++ is not a proper superset of C, so that backwards compatibility is not entirely maintained
    * Typically not a problem in practice...

- Note, certain C++ design goals conflict with modern techniques for:

  1. *Compiler optimization*
     - *e.g.*, pointers to arbitrary memory locations complicate register allocation and garbage collection
  2. *Software engineering*
     - *e.g.*, separate compilation complicates inlining due to difficulty of interprocedural analysis

Major C++ Enhancements

1. C++ supports object-oriented programming features
   - *e.g.*, single and multiple inheritance, abstract base classes, and virtual functions

2. C++ facilitates data abstraction and encapsulation that hides representations behind abstract interfaces
   - *e.g.*, the class mechanism and parameterized types

3. C++ provides enhanced error handling capabilities
   - *e.g.*, exception handling

4. C++ provides a means for identifying an object's type at runtime
   - *e.g.*, Run-Time Type Identification (RTTI)

Other Enhancements

- C++ enforces type checking via *function prototypes*

- Allows several different commenting styles

- Provides type-safe linkage

- Provides **inline** function expansion

- Built-in dynamic memory management via **new** and **delete** operators

- Default values for function parameters

- Operator and function overloading

Other Enhancements (cont'd)

- References provide "call-by-reference" parameter passing

- Declare constants with the **const** type qualifier

- New **mutable** type qualifier

- New **bool** boolean type

- New type-secure extensible I/O interface called *streams* and *iostreams*

Other Enhancements (cont'd)

- A new set of "function call"-style cast notations

- Variable declarations may occur anywhere statements may appear within a block

- The name of a **struct**, **class**, **enum**, or **union** is a type name

- Allows user-defined conversion operators

- Static data initializers may be arbitrary expressions

- C++ provides a namespace control mechanism for restricting the scope of classes, functions, and global objects

Language Features Not Part of C++

1. *Concurrency*

   • See "Concurrent C" by Nehrain Gehani

2. *Persistence*

   • See Exodus system and E programming language

3. *Garbage Collection*

   • See papers in USENIX C++ 1994

Function Prototypes

- C++ supports stronger type checking via *function prototypes*
  - Unlike ANSI-C, C++ *requires* prototypes for both function declarations and definitions
  - Function prototypes eliminate a class of common C errors
    - *e.g.*, mismatched or misnumbered parameter values and return types

- Prototypes are used for external declarations in header files, *e.g.*,

```
extern char *strdup (const char *s);
extern int strcmp (const char *s, const char *t);
FILE *fopen (const char *filename, const char *type);
extern void print_error_msg_and_die (const char *msg);
```

Function Prototypes (cont'd)

- Proper prototype use detects erroneous parameter passing at compile-time, *e.g.*,

```
#if defined (__STDC__) || defined (__cplusplus)
extern int freopen (const char *nm,
                    const char *tp,
                    FILE *s);
extern char *gets (char *);
extern int perror (const char *);
#else /* Original C-style syntax. */
extern int freopen (), perror ();
extern char *gets ();
#endif /* defined (__STDC__) */
/* ...*/
int main (void) {
    char buf[80];

    if (freopen ("./foo", "r", stdin) == 0)
        perror ("freopen"), exit (1);

    while (gets (buf) != 0)
        /* ...*/;
}
```

Function Prototypes (cont'd)

- The preceeding program fails mysteriously if the actual calls are:

```
/* Extra argument, also out-of-order! */
freopen (stdin, "newfile", 10, 'C');

/* Omitted arguments. */
freopen ("newfile", "r");
```

- A "Classic C" compiler would generally not detect erroneous parameter passing at compile time (though `lint` would)
  - Note, C++ **`lint`** utilities are not widely available, but running GNU **`g++ -Wall`** provides similar typechecking facilities

- Function prototypes are used in both *definitions* and *declarations*
  - Note, the function prototypes must be consistent!!!

Overloading

- Two or more functions or operators may be given the same name provided the *type signature* for each function is unique:

  1. *Unique argument types*:

     double square (**double**);
     Complex &square (Complex &);

  2. *Unique number of arguments*:

     void move (**int**);
     void move (**int**, **int**);

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

  – *e.g.*, the following declarations are ambiguous to the C++ compiler:

    extern double operator / (Complex &, Complex &);
    extern Complex **operator** / (Complex &, Complex &);

- Note, overloading is really just "syntactic sugar!"

C++ Classes

- The class is the basic protection and data abstraction unit in C++
  - *i.e.*, rather than "per-object" protection

- 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 both *built-in* and *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.*, **class**es, **struct**s, **union**s, **enum**s 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)

- A class is a "type constructor"
  - *e.g.*, in contrast to an Ada **package** or a Modula 2 **module**
    - Note, these are not types, they are "encapsulation units"
  - Until recently, C++ did not have a higher-level modularization mechanism...
    - This was a problem for large systems, due to lack of library management facilities and visibility controls
  - Recent versions of the ANSI C++ draft standard include mechanisms that addresses namespace control and visibility/scoping, *e.g.*,
    - Name spaces
    - Nested classes

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 a similar purpose to a C **struct**
  - However, it is extended to allow user-defined behavior, as well

Class Vector Example

- There are several significant limitations with built-in C and C++ arrays, *e.g.*,

  1. The size must be a compile-time constant, *e.g.*,

     ```
     void foo (int i)
     {
          int a[100], b[100]; // OK
          int c[i]; // Error!
     }
     ```

  2. Array size cannot vary at run-time

  3. Legal array bounds run from 0 to *size* $-$ 1

  4. No range checking performed at run-time, *e.g.*,

     ```
     {
          int a[10], i;
          for (i = 0; i <= 10; i++)
               a[i] = 0;
     }
     ```

  5. Cannot perform full array assignments, *e.g.*,

     ```
     a = b; // Error!
     ```

Class Vector Example (cont'd)

- We can write a C++ class to overcome some of these limitations, *i.e.*,
  - (1) compile-time constant size
  - (4) lack of range checking

- Later on we'll use inheritance to finish the job, *i.e.*,
  - (2) resizing
  - (3) non-zero lower bounds
  - (5) array assignment

Class Vector Example (cont'd)

- /* File Vector.h (this ADT is incomplete *wrt* initialization and assignment...!) */

```
#if !defined (_VECTOR_H) // Wrapper section
#define _VECTOR_H

typedef int T;
class Vector {
public:
     Vector (size_t len = 100) {
          this->size_ = len;
          this->buf_ = new T[len];
     }
     ~Vector (void) { delete [] this->buf_; }
     size_t size (void) const { return this->size_; }
     bool set (size_t i, T item);
     bool get (size_t i, T &item) const;

private:
     size_t size_;
     T *buf_;
     bool in_range (size_t i) const {
          return i >= 0 && i < this->size ();
     }
};
#endif /* _VECTOR_H */
```

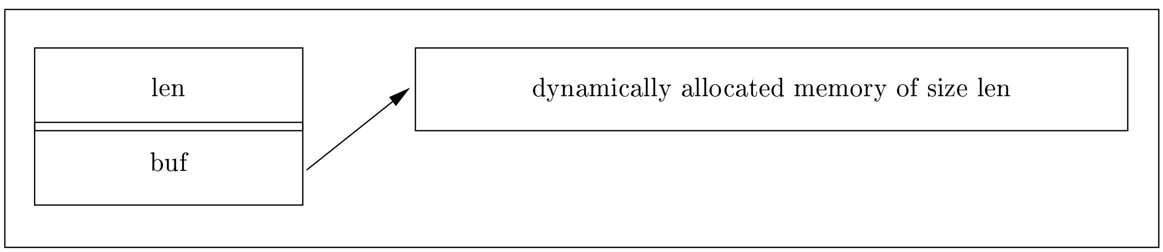
Class Vector Example (cont'd)

- /* File Vector.C */

```
#include "Vector.h"
bool Vector::set (size_t i, T item) {
     if (this->in_range (i)) {
          this->buf_[i] = item;
          return true;
     }
     else
          return false;
}

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

Class Vector Example (cont'd)



- The control block that represents an object of **class** Vector
  - Note, the control block may be allocated off the stack, the global data segment, or the heap
  - However, the **`buf`** field always points to memory allocated off the heap

Class Vector Example (cont'd)

- // File test.C

```
#include <libc.h>
#include "Vector.h"
void foo (size_t size) {
     Vector user_vec (size); // Call constructor
     int c_vec[size]; // Error, no dynamic range

     c_vec[0] = 0;
     user_vec.set (0, 0);

     for (int i = 1; i < user_vec.size (); i++) {
          int t;
          user_vec.get (i - 1, t);
          user_vec.set (i, t + 1);
          c_vec[i] = c_vec[i - 1] + 1;
     }

     // Error, private and protected data inaccessible
     size = user_vec.size_ - 1;
     user_vec.buf_[size] = 100;

     // Run-time error, index out of range
     if (user_vec.set (user_vec.size (), 1000) == false)
          err ("index out of range");

     // Index out of range not detected at runtime!
     c_vec[size] = 1000;

     // Destructor called when user_vec leaves scope
}
```

Class Vector Example (cont'd)

- Note that this example has several unnecessary limitations that are addressed by additional C++ features, *e.g.*,
  - `set/get` paradigm differs from C's built-in subscript notation
  - Error checking via return value is somewhat awkward
  - Only works for a vector of **int**s
  - Classes that inherit from Vector may not always want the extra overhead of range checking...
- The following example illustrates several more advanced C++ features
  - Don't worry, we'll cover these features in much greater detail over the course of the class!!!!

Class Vector Example (cont'd)

- /* File Vector.h */

```
// typedef int T;
template <class T>
class Vector {
public:
     struct RANGE_ERROR {};
     Vector (size_t len = 100): size_ (len) {
          if (this->size_ <= 0)
               throw Vector<T>::RANGE_ERROR ();
          else this->buf_ = new T[this->size_];
     }
     ~Vector (void) { delete [] this->buf_; }
     size_t size (void) const { return this->size_; }
     T &operator[] (size_t i) {
          if (this->in_range (i))
               return this->buf_[i];
          else throw Vector<T>::RANGE_ERROR ();
     }
protected:
     T &elem (size_t i) { return this->buf_[i]; }
private:
     size_t size_;
     T *buf_;
     bool in_range (size_t i) {
          return i >= 0 && i < this->size_;
     }
};
```

Class Vector Example (cont'd)

- // File test.C

```
#include <libc.h>
#include "Vector.h"
void foo (size_t size) {
    try { // Illustrates exception handling...
        Vector<int> user_vec (size); // Call constructor
        int c_vec[size]; // Error, no dynamic range

        c_vec[0] = user_vec[0] = 0;

        for (int i = 1; i < user_vec.size (); i++) {
            user_vec[i] = user_vec[i - 1] + 1;
            c_vec[i] = c_vec[i - 1] + 1;
        }

        // Error, private and protected data inaccessible
        size = user_vec.size_ - 1;
        user_vec.buf_[size] = 100;
        user_vec.elem (3) = 120;

        // Run-time error, RANGE_ERROR thrown
        user_vec[user_vec.size ()] = 1000;

        // Index out of range not detected at runtime!
        c_vec[size] = 1000;

        // Destructor called when user_vec leaves scope
    }
    catch (Vector<int>::RANGE_ERROR) { /* ...*/ }
}
```

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*

- As with C, the life-time of a C++ object is either *static*, *automatic*, or *dynamic*
  - C and C++ refer 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"

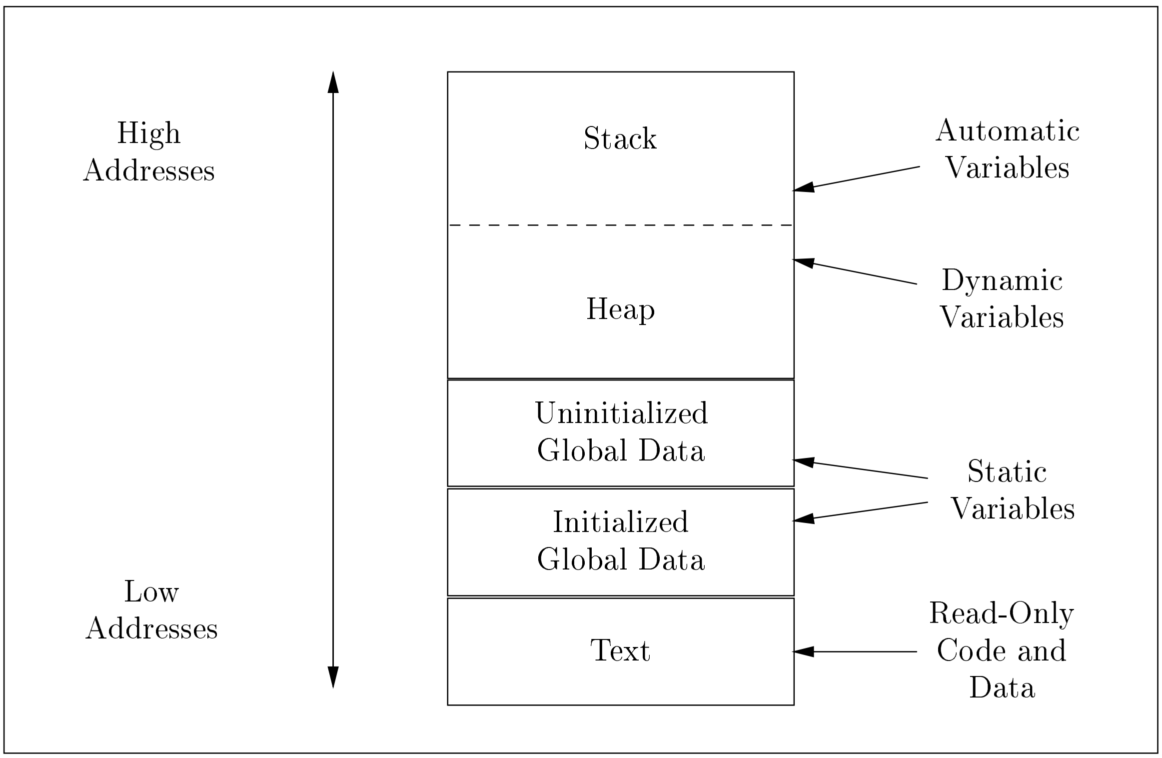
  3. *Dynamic*

     – *i.e.*, it lives between corresponding calls to operators **new** and **delete**

       ∗ Or `malloc` and `free`

     – Dynamic objects have life-times that extend beyond their original scope

C++ Objects (cont'd)



- Typical layout of memory objects in the process address space

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...

C++ Objects (cont'd)

- Several workarounds exist, however, *e.g.*,
  - Use Eiffel or LISP ;-)
  - Use inheritance to derive from base **class** Collectible
    - However, this only works then for a subset of classes (*i.e.*, doesn't work for built-in types. . .)
  - Use the class-specific **new** and **delete** operators to define a memory management facility using reference counts to reclaim unused memory
  - Adapt Hans Boehm's conservative garbage collector for C to C++. . .
- No solution is optimal, however, so storage management is often performed "by hand" (ugh ;-))

C++ Object-Oriented Features

- C++ provides three characteristics generally associated with object-oriented programming:

  1. *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*

  2. *Single and Multiple Inheritance*

     – A derived class inherits operations and attributes from one or more base classes, possibly providing additional operations and/or attributes

  3. *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 are instantiated at compile-time

C++ Object-Oriented Features (cont'd)

- C++'s object-oriented features encourage designs that

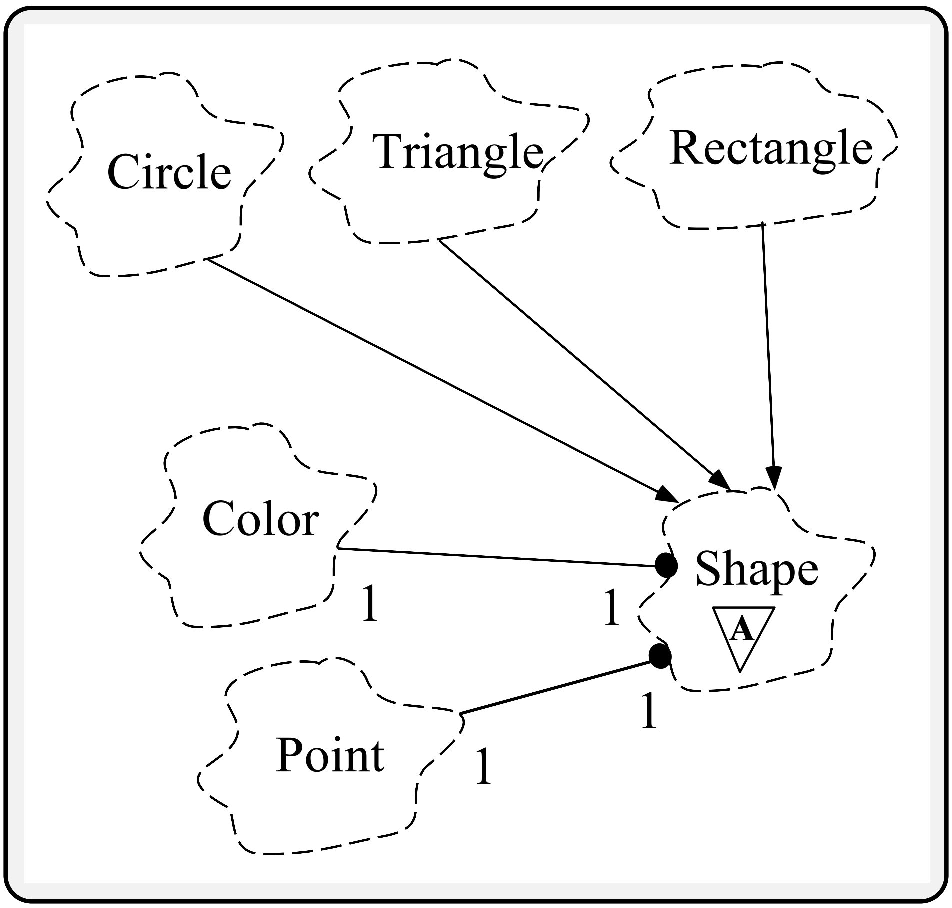
  1. Explicitly distinguish *general properties* of related concepts from

  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

C++ Object-Oriented Features (cont'd)



- Note, the "OOD challenge" is to map arbitrarily complex system architectures into inheritance hierarchies

C++ Object-Oriented Features (cont'd)

- Inheritance and dynamic binding facilitate the construction of "program families" and frameworks
  - Program families are sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members
  - A framework is an integrated set of components that collaborate to product a reuseable architecture for a family of related applications
- It also supports the *open/closed* principle
  - *i.e.*, *open* with respect to extensibility, *closed* with respect to stability

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.*,

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

Inheritance Example: Ada-style Vectors

- /* File Ada_Vector.h (still incomplete wrt assignment and initialization) */

```
#if !defined (_ADA_VECTOR_H)
#define _ADA_VECTOR_H
#include "Vector.h"
template <class T>
class Ada_Vector : private Vector<T>
{
public:
        Ada_Vector (int l, int h);
        T &operator() (int i);

        // extend visibility from class Vector
        Vector::size;
        Vector::RANGE_ERROR;
        // Note, destructor is not inherited...
private:
        int lo_bnd_;
};
#endif /* _ADA_VECTOR_H */
```

Inheritance Example: Ada-style Vectors (cont'd)

- /* File Ada_Vector.C */

```
template <class T>
Ada_Vector<T>::Ada_Vector (int l, int h)
    : lo_bnd_ (l), Vector<T> (h - l + 1) {}

template <class T>
T &Ada_Vector<T>::operator() (int i) {
    if (this->in_range (i - this->lo_bnd_))
    // Call inherited operation, no range checking
        return this->elem (i - this->lo_bnd_);
    else
        throw Ada_Vector<T>::RANGE_ERROR ();
    /* or
    (*this)[i - this->lo_bnd_]; */
}
```

Inheritance Example: Ada-style Vectors (cont'd)

- Example Ada Vector Usage
  - // File main.C

```
#include <stream.h>
#include "Ada_Vector.h"
extern "C" int atoi (const char *);

int main (int argc, char *argv[]) {
     try {
          int lower = atoi (argv[1]);
          int upper = atoi (argv[2]);
          Ada_Vector<int> ada_vec (lower, upper);

          ada_vec (lower) = 0;

          for (int i = lower + 1; i <= ada_vec.size (); i++)
               ada_vec (i) = ada_vec (i - 1) + 1;

          // Run-time error, index out of range
          ada_vec (upper + 1) = 100;

          // Vector destructor called when
          // ada_vec goes out of scope
     }
     catch (Ada_Vector<int>::RANGE_ERROR) { /* ...*
}
```

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
    * Virtual functions introduce a small amount of overhead for each virtual function call

Dynamic Binding Preview (cont'd)

- *e.g.*,

```
struct X { /* Base class */
     int f (void) { puts ("X::f"); } // Non-virtual
     virtual int vf (void) { puts ("X::vf"); } // Virtual
};
struct Y : public X { /* Derived class */
     int f (void) { puts ("Y::f"); } // Non-virtual
     virtual int vf (void) { puts ("Y::vf"); } // Virtual
};

void foo (X *x) { /* Note, can also use references...*/
     x->f (); /* direct call: _f_1X (x); */
     x->vf (); /* indirect call: (*x->vptr[1]) (x) */
}

int main (void) {
     X x;
     Y y;
     foo (&x); // X::f, X::vf
     foo (&y); // X::f, Y::vf
}
```

Dynamic Binding Preview (cont'd)

- Each class with 1 or more **virtual** functions generates one or more virtual tables (*vtables*)
  - Note, multiple inheritance creates multiple *vtables*

- A *vtable* is *logically* an array of pointers to methods
  - A *vtable* is typically implemented as an array of pointers to C functions

- Each object of a class with virtual functions contains one or more virtual pointers (*vptrs*), which point at the appropriate *vtable* for the object
  - The constructor automatically assigns the *vptrs* to point to the appropriate *vtable*

New-Style 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 new "continue until end-of-line" comment style, *e.g.*,

     ```
     // This is a single-line C++ comment
     ```

- Note, C-style comments do not nest

  ```
  /*
  /* Hello world program */
  int main (void) {
       printf ("hello world\n");
  }
  */
  ```

- However, these two styles nest, so it is possible to comment out code containing other comments, *e.g.*,

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

New-Style Comments (cont'd)

- Naturally, it is still possible to use C/C++ preprocessor directives to comment out blocks of code:

```
#if 0
/* Make sure only valid C++ code goes here! */
/* i.e., don't use apostrophes! */
#endif
```

- Beware of subtle whitespace issues...

```
int b = a //* divided by 4 */4;
-a;
/* C++ preprocessing and parsing. */
int b = a -a;
/* C preprocessing and parsing. */
int b = a/4; -a;
```

- Note, in general it is best to use whitespace around operators and other syntactic elements, *e.g.*,

```
char *x;
int foo (char * = x); // OK
int bar (char*=x); // Error
```

Type-Safe Linkage

- Type-safe linkage allows the linker to detect when a function is declared and/or used inconsistently, *e.g.*,:

```
// File abs.c
long abs (long arg)
{
    return arg < 0 ? -arg : arg;
}

// File application.c
#include <stdio.h>
int abs (int arg);
int main (void) { printf ("%d\n", abs (-1)); }
```

- Without type-safe linkage, this error would remain hidden until the application was ported to a machine where **int**s and **long**s were different sizes
  - *e.g.*, Intel 80286

Type-Safe Linkage (cont'd)

- Type-safe linkage encodes all C++ function names with the types of their arguments (a.k.a. "name mangling"!)

- *e.g.*,

  long abs (**long** arg) → _abs__Fl
  int abs (**int** arg) → _abs__Fi

- Therefore, the linker may be used to detect mismatches between function prototypes, function definitions, and function usage

Type-Safe Linkage (cont'd)

- Name mangling was originally created to support overload resolution

- Only function names are mangled
  - *i.e.*, variables, constants, enums, and types are not mangled...

- On older C++ compilers, diagnostic messages from the linker are sometimes rather cryptic!
  - See the `c++filt` program...

Typesafe Linkage (cont'd)

- Language interoperability issues
  - This problem arises as a side effect of using type-safe linkage in C++
  - C functions used in C++ code (*e.g.*, standard UNIX library functions) must be explicitly declared as requiring C linkage (*i.e.*, names are not mangled) via the new **extern** "C" declaration

- *e.g.*,

```
extern "C" int abs (int i);
double abs (double d);
Complex abs (Complex &c);

int foo (int bar) {
     cout << abs (Complex (-10, 4.5));
          // calls _abs__F7Complex
          << abs (bar) // calls _abs
          << abs (3.1416) // calls _abs__Fd

}
```

Typesafe Linkage (cont'd)

- Language interoperability issues (cont'd)
  - Other syntactic forms of **extern** blocks:

```
extern "C" {
     char *mktemp (const char *);
     char *getenv (const char *);
}
```

  - Encapsulating existing header files

```
#if defined (__cplusplus)
extern "C" {
#endif /* __cplusplus */
#include <string.h>
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

  - Note, **extern** blocks also support other languages...
    - *e.g.*, FORTRAN, Pascal, Ada, etc.

Inline Functions

- Many programming languages force developers to choose between:

  1. Modularity/abstraction (function call)

  2. Performance (macro or inline-expansion by-hand)

- C++ allows inline function expansion, which has several advantages:

  1. It combines the efficiency of a macro with the type-security and abstraction of a function call

  2. It reduces *both* execution time and code size (potentially)

  3. It discourages the traditional reliance upon macro preprocessor statements

Inline Functions (cont'd)

- Here's an example of a common C problem with the preprocessor:
  - Classic C macro, no sanity-checking at macro expansion time

    ```
    #define SQUARE(X) ((X) * (X))
    int a = 10;

    int b = SQUARE (a++); // trouble!!! (a++) * (a++)
    ```

  - C++ inline function template

    ```
    template<class T> inline
    T square (T x) { return x * x; }

    int c = square (a++); // OK
    ```

Inline Functions (cont'd)

- Points to consider about inline functions:

  1. Class methods that are defined in their declaration are automatically expanded inline

  2. It is difficult to debug code where functions have been inline expanded and/or optimized

  3. Compilers require more time and space to compile when there are many inline functions

  4. Inline functions do not have the *pseudo-polymorphic* properties of macros

     – However, **inline** templates approximate this functionality

  5. Compilers often have limits on the size and type of function that can be inlined.

     – *e.g.*, if stack frame is very large:

     ```
     int foo (void) {
          int local_array[1000000];
          // ...
     ```

     – This can cause surprising results *wrt* code size, *e.g.*,

     ```
     int bar (void) { foo (); foo (); }
     ```

Inline Functions (cont'd)

- As an example of inlining in C++, we will discuss a simple run-time function call "trace" facility

  - Provides a rudimentary debugging facility

    * *e.g.*, useful for long-running network servers

- The goals are to be able to:

  1. Determine the dynamic function calling behavior of the program, *i.e.*, "tracing"

  2. Allow for fine-grain control over whether tracing is enabled, *e.g.*,

     - At compile-time (remove all traces of Trace and incur no run-time penalty)

     - At run-time (via signals and/or command-line options)

  3. Make it easy to automate source code instrumentation

- *e.g.*, write a regular expression to match function definitions and then insert code automatically

Inline Functions (cont'd)

- Example output:

```
% CC -D__INLINE__ main.C trace.C
% a.out 10 1
enter int main (int argc, char *argv[]) in file main.C on line 25
 enter void foo (void) (file main.C, line 8)
  enter void foo (void) in (file main.C, line 8)
   enter void foo (void) in (file main.C, line 8)
    enter void foo (void) in (file main.C, line 8)
     enter void foo (void) in (file main.C, line 8)
      enter void foo (void) in (file main.C, line 8)
       enter void foo (void) in (file main.C, line 8)
        enter void foo (void) in (file main.C, line 8)
         enter void foo (void) in (file main.C, line 8)
          enter void foo (void) in (file main.C, line 8)
           enter void foo (void) in (file main.C, line 8)
           leave void foo (void)
          leave void foo (void)
         leave void foo (void)
        leave void foo (void)
       leave void foo (void)
      leave void foo (void)
     leave void foo (void)
    leave void foo (void)
   leave void foo (void)
  leave void foo (void)
 leave void foo (void)
leave int main (int argc, char *argv[])
```

Inline Functions (cont'd)

- *e.g.*, main.C

```
#include "Trace.h"

void foo (int max_depth) {
     T ("void foo (void)");
     /* Trace __ ("void foo (void)", 8, "main.c") */
     if (max_depth > 0) foo (max_depth − 1);
     /* Destructor called automatically */
}

int main (int argc, char *argv[]) {
     const int MAX_DEPTH =
          argc == 1 ? 10 : atoi (argv[1]);
     if (argc > 2)
          Trace::set_nesting_indent (atoi (argv[2]));
     if (argc > 3)
          Trace::stop_tracing ();
     T ("int main (int argc, char *argv[])");
     foo (MAX_DEPTH);
     return 0;
     /* Destructor called automatically */
}
```

Inline Functions (cont'd)

- // Trace.h

```
#if !defined (_TRACE_H)
#define _TRACE_H
#if defined (NTRACE) // compile-time omission
#define T(X)
#else
#define T(X) Trace __ (X, __LINE__, __FILE__)
#endif /* NTRACE */

class Trace {
public:
     Trace (char *n, int line = 0, char *file = "");
     ~Trace (void);
     static void start_tracing (void);
     static void stop_tracing (void);
     static int set_nesting_indent (int indent);
private:
     static int nesting_depth_;
     static int nesting_indent_;
     static int enable_tracing_;
     char *name_;
};
#if defined (__INLINE__)
#define INLINE inline
#include "Trace.i"
#else
#define INLINE
#endif /* __INLINE__ */
#endif /* _TRACE_H */
```

Inline Functions (cont'd)

- *e.g.*, /* Trace.i */

```
#include <stdio.h>
INLINE
Trace::Trace (char *n, int line, char *file) {
    if (Trace::enable_tracing_)
        fprintf (stderr, "%*senter %s (file %s, line %d)\n",
            Trace::nesting_indent_ *
            Trace::nesting_depth_++,
            "", this->name_ = n, file, line);
}

INLINE
Trace::~Trace (void) {
    if (Trace::enable_tracing_)
        fprintf (stderr, "%*sleave %s\n",
            Trace::nesting_indent_ *
            --Trace::nesting_depth_,
            "", this->name_);
}
```

Inline Functions (cont'd)

- *e.g.*, /* Trace.C */

```
#include "Trace.h"

#if !defined (__INLINE__)
#include "Trace.i"
#endif /* __INLINE__ */

/* Static initializations */
int Trace::nesting_depth_ = 0;
int Trace::nesting_indent_ = 3;
int Trace::enable_tracing_ = 1;

void Trace::start_tracing (void)
      Trace::enable_tracing_ = 1;
}
void Trace::stop_tracing (void) {
      Trace::enable_tracing_ = 0;
}
int Trace::set_nesting_indent (int indent) {
      int result = Trace::nesting_indent_;
      Trace::nesting_indent_ = indent;
      return result;
}
```

Dynamic Memory Management

- Dynamic memory management is now a built-in language construct, *e.g.*,

  – Traditional C-style

    ```
    void *malloc (size_t);
    void free (void *);
    // ...
    int *a = malloc (10 * sizeof *a);
    free ((void *) a);
    ```

  – C++ syntax

    ```
    int *a = new int[10];
    int *b = new int;
    // ...
    delete [] a;
    delete b;
    ```

- Built-in support for memory management improves:

  1. *Type-security*
  2. *Extensibility*
  3. *Efficiency*

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
- *e.g.*,

```
const char *foo = "on a clear day";
char *const bar = "you can C forever!";
const char *const zippy = "yow!";

foo = "To C or not to C?" // OK
foo[7] = 'C'; // error, read-only location

// error, can't assign to const pointer bar
bar = "avoid cliches like the plague.";
// OK, but be careful of read-only memory!!!
bar[1] = 'D';

const int index = 4 - 3; // index == 1
// read-only an array of constant ints
const int array[index + 3] = {2, 4, 8, 16};
```

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.*,

    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

Const Type Qualifier (cont'd)

- **const** methods
  - **const** methods may specify that certain read-only operations take place on user-defined **const** objects, *e.g.*,

```
class String {
public:
     size_t size (void) const { return this->len_; }
     void set (size_t index, char new_char);
     // ...
private:
     size_t len;
};

const String string_constant ("hello");
string_constant.size (); // Fine
```

  - A **const** method may not directly modify its **this** pointer

```
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

Stream I/O (cont'd)

- The stream and iostream class categories replace `stdin`, `stdout`, and `stderr` with `cout`, `cin`, and `cerr`

- These classes may be used by overloading the << and >> operators

  – C++ does not get a segmentation fault since the "correct" function is called

```
#include <iostream.h>
char *name = "joe";
int id = 1000;
cout << "name = " << name << ", id = " << id << '\n';
// cout.operator<< ("name = ").operator<< ("joe")...
```

  – In contrast, old C-style I/O offers no protection from mistakes, and gets a segmentation fault on most systems!

```
printf ("name = %s, id = %s\n", name, id);
```

Stream I/O (cont'd)

- Be careful using Stream I/O in constructors and destructors for global or static objects, due to undefined linking order and elaboration problems...

- In addition, the Stream I/O approach does not work particularly well in a multi-threaded environment...
  - This is addressed in newer compilers that offer thread-safe iostream implementations

Boolean Type

- C++ has added a new build-in type called **bool**
  - 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
  - Statements such as **if** and **while** are now converted to **bool**
  - All operators that conceptually return truth values return **bool**
    * *e.g.*, the operands of &&, ||, and !, but not &, |, and ~

References

- C++ allows *references*, which may be:
  1. Function parameters
  2. Function return values
  3. Other objects

- A reference variable creates an alternative name (a.k.a. "alias") for an object

- References may be used instead of pointers to facilitate:
  1. Increased code clarity
  2. Reduced parameter passing costs
  3. Better compiler optimizations

- References use *call-by-value* syntax, but possess *call-by-reference* semantics

References (cont'd)

- *e.g.*, consider a `swap` abstraction:

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

int main (void) {
     int a = 10, b = 20;
     printf ("a = %d, b = %d\n", a, b);
     swap (a, b);
     printf ("a = %d, b = %d\n", a, b);
}
```

- There are several problems with this code

  1. It doesn't swap!

  2. It requires a function call

  3. It only works for integers!

References (cont'd)

- *e.g.*, swap

```
void swap (int *xp, int *yp) {
     int t = *xp; *xp = *yp; *yp = t;
}
int main (void) {
     int a = 10, b = 20;
     printf ("a = %d, b = %d\n", a, b);
     swap (&a, &b);
     printf ("a = %d, b = %d\n", a, b);
}

#define SWAP(X,Y,T) \
     do {T __ = (X); (X) = (Y); (Y) = __;} while (0)
int main (void) {
     int a = 10, b = 20;
     printf ("a = %d, b = %d\n", a, b);
     SWAP (a, b, int); // beware of a++!
     printf ("a = %d, b = %d\n", a, b);
}
```

References (cont'd)

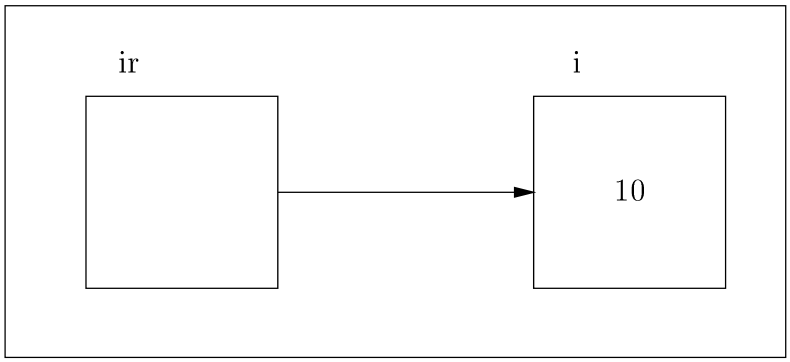
- *e.g.*, swap

```
template <class T> inline void
swap (T &x, T &y) {
     T t = x;
     x = y;
     y = t;
}

int main (void) {
     int a = 10, b = 20;
     double d = 10.0, e = 20.0;

     printf ("a = %d, b = %d\n", a, b);
     printf ("d = %f, e = %e\n", d, e);
     swap (a, b);
     swap (d, e);
     printf ("a = %d, b = %d\n", a, b);
     printf ("d = %f, e = %e\n", d, e);
}
```

References (cont'd)



- With references (as with classes), it is important to distinguish *initialization* from *assignment*, *e.g.*,

```
int i = 10;
// initialization of ir
int &ir = i; // Equivalent to int *const ip = &i;
ir = ir + 10; // dereference is automatically done
              // *ip = *ip + 10;
```

- Once initialized, a reference cannot be changed
  - *i.e.*, it may not be reassigned to reference a new location
  - Note, after initialization all operations affect the referenced *object*
    - *i.e.*, *not* the underlying **const** pointer...

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 Pascal, *e.g.*,

```
// function prototype from math.h library
extern double log10 (double param);

if ((int) log10 ((double) 7734) != 0)
     ; /* C style type cast notation */

if (int (log10 (double (7734))) != 7734)
     ; // C++ function-style cast notation
```

- This "function call" is performed at compile time

Type Cast Syntax (cont'd)

- This type cast syntax is also used to specify explicit type conversion in the C++ class mechanism
  - This allows multiple-argument casts, *i.e.*, "constructors"

- *e.g.*,:

```
class Complex {
public:
      Complex (double, double = 0.0);
      // ...
private:
      double real, imaginary;
};

// Convert 10.0 and 3.1416 into a Complex object
Complex c = Complex (10.0, 3.1416);

// Note that old-style C syntax would not suffice here...
Complex c = (Complex) (10.0, 3.1416);
```

Type Cast Syntax (cont'd)

- Note, there are a variety of syntactical methods for constructing objects in C++, *e.g.*,

   1. Complex c1 = 10.0;

   2. Complex c2 = (Complex) 10.0;

   3. Complex c3 = Complex (10.0);

   4. Complex c4 (10.0);

- I recommend version 4 since it is the most consistent and also works with built-in types...

   – It also generalizes to multiple-argument casts...

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 (char *name, char *grade = "A");
    // additional declarations and definitions...

    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 (and give them default values)

Default Parameters (cont'd)

- Default parameter passing semantics are similar to those in languages like Ada:
  - *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.*,

  ```
  extern 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; )
        /* ...*/;
}
```

Declaration Statements (cont'd)

- The following example illustrates declaration statements and also shows the use of the "scope resolution" operator

```
#include <iostream.h>
struct Foo { static int var; };
int Foo::var = 20;
const int MAX_SIZE = 100;
int var = 10;

int main (void) {
    int k;
    k = call_something ();
    // Note the use of the "scope resolution" operator
    // (::) to access the global variable var
    int var = ::var - k + Foo::var;

    for (int i = var; i < MAX_SIZE; i++)
        for (int j = 0; j < MAX_SIZE; j++) {
                int k = i * j;
                cout << k;
                double var = k + ::var * 10.4;
                cout << var;
        }
}
```

Declaration Statements (cont'd)

- However, the declaration statement feature may encourage rather obscure code since the scoping rules are not always intuitive or desirable

- Note, new features in ANSI C++ allow definitions in the **switch**, **while**, and **if** condition expressions...

- According to the latest version of the ANSI/ISO C++ draft standard, the scope of the definition of `i` in the following loop is limited to the body of the **for** loop:

```
{
    for (int i = 0; i < 10; i++)
        /* ...*/;
    for (int i = 0; i < 20; i++)
        /* ...*/;
}
```

Abbreviated Type Names

- Unlike C, C++ allows direct use of user-defined type tag names, without requiring a preceding **union**, **struct**, **class**, or **enum** specifier, *e.g.*,

```
struct Tree_Node { /* C code */
     int item_;
     struct Tree_Node *l_child_, *_child_;
};

struct Tree_Node { /* C++ code */
     int item_;
     Tree_Node *l_child_, *r_child_;
}
```

- Another way of looking this is to say that C++ automatically **typedef**s tag names, *e.g.*,

```
typedef struct Tree_Node Tree_Node;
```

- Note, this C++ feature is incompatible with certain Classic and ANSI C identifier naming conventions, *e.g.*,

```
struct Bar { /* ...*/ };
struct Foo { };
typedef struct Foo Bar; // Illegal C++, legal C!
```

User-Defined Conversions

- The motivation for user-defined conversions are similar to those for operator and function overloading
  - *e.g.*, reduces "tedious" redundancy in source code
  - However, both approaches have similar problems with readability...

- User-defined conversions allow for more natural looking mixed-mode arithmetic for user-defined types, *e.g.*,:

```
Complex a = Complex (1.0);
Complex b = 1.0; // implicit 1.0 -> Complex (1.0)

a = b + Complex (2.5);
a = b + 2.5 // implicit 2.5 -> Complex (2.5)

String s = a; // implicit a.operator String ()
```

User-Defined Conversions (cont'd)

- Conversions come in two flavors:

  1. *Constructor Conversions*:
     - Create a new object from objects of existing types
  2. *Conversion Operators*:
     - Convert an existing object into an object of another type

- *e.g.*,

```
class Complex {
public:
    Complex (double); // convert double to Complex
    operator String (); // convert Complex to String
    // ...
};
int foo (Complex c) {
    c = 10.0; // c = Complex (10.0);
    String s = c; // c.operator String ();
    cout << s;
}
```

User-Defined Conversions (cont'd)

- In certain cases, the compiler will try a single level of user-defined conversion to determine if a type-signature matches a particular use, *e.g.*,

```
class String {
public:
     String (const char *s);
     String &operator += (const String &);
};
String s;
s += "hello"; // s += String ("hello");
```

- Note, it is easy to make a big mess by abusing the user-defined conversion language feature...
  - Especially when conversions are combine with templates, inheritance virtual functions, and overloading, etc.

Static Initialization

- In C, all initialization of static objects must use constant expressions, *e.g.*,:

```
int i = 10 + 20; /* file scope */
int foo (void) {
     static int j = 100 * 2 + 1; /* local scope */
}
```

- However, static initializers can be comprised of arbitrary C++ expressions, *e.g.*,

```
extern int foo (void); // file scope
int a = 100;
int i = 10 + foo ();
int j = i + *new int (1);

int foo (void) {
     static int k = foo ();
     return 100 + a;
}
```

- Note, needless to say, this can become rather cryptic, and the order of initialization is not well defined between modules

Miscellaneous Differences

- In C++, **sizeof** ('a') == **sizeof** (**char**); in C, **sizeof** ('a') == **sizeof** (**int**)
  - This facilitates more precise overloading...

- **char** str[5] = "hello" is valid C, but C++ gives error because initializer is too long (because of hidden trailing '\0')

- In C++, a function declaration **int** f(); means that f takes no arguments (same as **int** f(**void**);). In C it means that f can take any number of arguments of any type at all!
  - C++ would use **int** f (...);

- In C++, a class may not have the same name as a **typedef** declared to refer to a different type in the same scope

Miscellaneous Differences (cont'd)

- In C++, a **struct** or **class** is a scope; in C a **struct**, **enum**, or **enum** literal are exported into the "global scope," *e.g.*,

```
struct Foo { enum Bar {BAZ, FOOBAR, BIZBUZZ}; };
/* Valid C, invalid C++ */
enum Bar bar = BAZ;
// Valid C++, invalid C
Foo::Bar bar = Foo::BAZ;
```

- The type of an **enum** literal is the type of its enumeration in C++; in C it is an **int**, *e.g.*,

```
/* True in C, not necessarily true in C++. */
sizeof BAZ == sizeof (int);
/* True in C++, not necessarily true in C. */
sizeof Foo::BAZ == sizeof (Foo::Bar);
```

Miscellaneous Differences (cont'd)

- In ANSI C, a global **const** has external linkage by default; in C++ it has internal linkage, *e.g.*,

```
/* In C++, "global1" is not visible to other modules. */
const int global1 = 10;
/* Adding extern makes it visible to other modules. */
extern const int global2 = 100;
```

- In ANSI C, a **void** * may be used as the right-hand operand of an assignment or initialization to a variable of any pointer type, whereas in C++ it may not (without using a cast...)

```
void *malloc (size_t);
/* Valid C, invalid C++ */
int *i = malloc (10 * sizeof *i);
/* Valid C, valid C++ */
int *i = (int *) malloc (10 * sizeof *i);
```

Summary

- C++ adds many, many, many new features to the C programming language

- It is not necessary to use all the features in order to write efficient, understandable, portable applications

- C++ is a "moving target"

  – Therefore, the set of features continues to change and evolve

  – However, there are a core set of features that are important to understand and are ubiquitous