C++

- Language basics - comparison to Java
- Operator overloading
- Pointers versus references
- Dynamic dispatch
- Visibility
C++

• An object-oriented language built on C in mid-1980’s
• Multiple inheritance
  – Operator overloading - use of existing operators with new types or arguments
• Objects- on a stack frame versus in the heap
• Use of pointers and reference variables
  – Use pointers to refer to dynamically created objects, not references
• Choice of direct or dynamic dispatch (virtual functions)
C++ versus Java

- Pointers to objects
- Multiple inheritance
- Objects can be created statically or dynamically
- Virtual functions dynamically bound
- Operator overloading
- Class implementation can be defined separately from class specification
- C syntax
- Allows global procedure definition

- References to objects (more restricted than pointers)
- Single inheritance
- All objects created dynamically
- All functions dynamically bound
- No operator overloading
- Class implementation with class specification
- C-like syntax
- All procedures and functions associated with a class
Example

class X //this class to demonstrate automatic // calls constructor and destructor
{
public:
    X() { printf("Constructor called.\n"); }
    ~X() { printf("Destructeur called.\n"); }
    void f(int i=0)
    {
        cout << "i is " << i << "\n"; 
    }
} x2; //this declares an object x of type X

main()
{
    if (1) { x1; x1.f(); }
}

Output
Constructor called
Constructor called
i is 0
Destructor called
Destructor called.
Subtyping and Derived Classes

• A derived class inherits some behavior from its base class
  
  \[
  \text{class} \ V : \ \text{public} \ A
  \]

• A **subtype** value can be used anywhere its supertype value can be used.

• If a public derived class inherits **all** members from its base class, without overriding any, then it has a **subtyping relation** with its base class.
#include <stdio.h>
#include <stream.h>

class vector
{
    int sz;
    int *v;

    public: vector(int);
    constructor
    ~vector() { delete v; }
    destructor
    int size() const { return sz; }
    int& elem(int i) const
    { return v[i]; }
    int& operator[](int);
};

C++ allows overloading of operators:
e.g., subscript operator
void error(char *s)  
{  cout << s << "\n";  
   exit(1);  
}

vector::vector(int i)  
{  if (i <= 0) error("bad vector size");  
   sz = i;  
   v = new int[i];  
}

int& vector::operator[](int i)  
{  if (i < 0 || i >= sz) error("index out of bounds");  
   return v[i];  
}

procedure
constructor code
overloaded subscript operator code
does bounds check so is safe!  

elem() is not safe.
References in C++

- References cannot be null; a reference designates a particular object
- Once a reference is given a value, it cannot be changed to point to a different object
- There is no explicit way to access the value at the memory address associated with a reference; you use it like you would use a variable.
- Used to simulate call by reference in C++ and to return values from functions
Implementation Reuse through Inheritance

class vec : public vector
{ int high, low; //private members of class
  public: vec(int, int);
    int& elem(int i){return vector::elem(i - low);}
    int& operator[](int);
};
vec::vec(int i, int j) : vector(j - i + 1)
{ if (j < i) j = i;
  low = i;//initialize locals
  high = j;
}
int& vec::operator[](int i)
{ if (i < low || i > high)
    error("index out of bounds for vec");
  return elem(i);}

Explicit call to base class constructor with args;
Initializes superclass

Continues same C++ program
Multiple Constructors

class newvec : public vector
{
public:
    newvec(int s) : vector (s) {}
    newvec(const newvec&);
    ~newvec() {}
    void operator=(const newvec&);
    newvec operator+(const newvec&) const;
};
//define a second constructor to create a newvec
//initialized to be a copy of another newvec
newvec::newvec(const newvec& a) : vector(a.size())
{
    for (int i = 0; i < a.size(); i++)
        elem(i) = a.elem(i);
}
Signature

• The number and types of parameters and return value

int foo(int x, bar *){…}
signature int(int, bar*)
Overloaded Operators

//define an assignment operator
void newvec::operator=(const newvec& a)
{
    int i;
    if (size() != a.size())
        error("bad vector size for =");
    for (i = 0; i < size(); i++) elem(i) = a.elem(i);
}

newvec newvec::operator+(const newvec& b) const
//infix addition operator
{
    int sz = size(); int i;
    if (sz != b.size()) error("bad vector sizes for +");
    newvec sum(sz);
    for (i = 0; i < sz; i++)
        sum.elem(i) = this->elem(i) + b.elem(i);
    return sum;
}
Inheritance Example, cont.

```c++
main()
{
    int i;
    newvec v1(10);
    newvec v2(20);
    for (i = 0; i < 10; i++) v1[i] = i;
    for (i = 0; i < 20; i++) v2[i] = i-1;
    newvec v3(v1);
    newvec v4(v2);
    newvec v5 = v1 + v3; // shows overloaded = and +
    for (i = 0; i < 10; i++) cout << v5.elem(i) << " ";
    cout << "\n";
}
// run by typing: g++ bgrvec.cc followed by
// > a.out
// 0 2 4 6 8 10 12 14 16 18
```
Hierarchy

vector
  sz, v[
  vector(), ~vector(), elem(), op[

vec
  high, low
  elem(), op[

newvec
  newvec(), newvec(&vec),
  op=, op +
Visibility in C++

• For members and member functions
  – **Private** - only visible within that class
  – **Protected** - only visible within that class and its derived classes
  – **Public** - visible to everyone

• Always use public inheritance! Private inheritance exists but is difficult to use.

```cpp
class D : public B
class D : private W
```

- Can use public interface of D on B objects; Cannot use any D methods on W objects.
How is inheritance useful?

- Reuse by inheriting implementation, modifying and/or extending it
  - E.g., `vector` and `newvec` (inherit); `vector` and `vec` (extending)

- Allows enforcement of same public interface to objects in all subclasses (same method name and signature even if they do different things in their implementation)
  - E.g., `elem()` in `vector` and `vec`
How is Inheritance useful?

• Can create superclasses by factoring out common data and actions from subclasses
  – E.g., putting size in vector
  – Facilitates changes by localizing effects
  – Can avoid clerical errors if meaning of a same-named entity changes
Object Creation

Dynamic: \texttt{vector *v = new vector(10);} \\
\hspace{1cm} \texttt{vec *w = new vec(10);} \\

Static: \texttt{vector a(10);} \\
\hspace{1cm} \texttt{vec b(10);} \\

Differences - object created dynamically, with a pointer, in C++ are stored on heap; otherwise, they are stored on a stack frame

\textbf{BE CAREFUL:} \texttt{v = w;} \hspace{1cm} \texttt{a = b;} are legal but with different effects. \texttt{v=w} makes \texttt{v} point to a \texttt{vec} object; \texttt{a=b} truncates the \texttt{vec} object data fields which don’t belong to \texttt{vector} to fit in the stack storage!

\textbf{Always create objects using pointers in C++}
Virtual Functions

• Dynamic binding only happens in C++ with functions declared to be *virtual*.

  Define `void vector::printv(){..}`
  ```cpp
  void vec::printv(){..}
  ```

  Declare `vector *v, vector *vv, vec *w`

  Initialize `v, vv, w` to point to objects of their declared types

  Execute `vv = w; v->printv(); vv->printv();`

  Both calls will execute `vector::printv()`

  **Choice of function based on declared type of pointer because methods NOT declared to be virtual!**
Virtual Fcns, Example

```c++
#include <stdio.h>
#include <stream.h> //example inspired by pohl book

class B {
public:
  virtual void print_i() {cout << 1 <<
                           " inside B\n";}
};

class D : public B {
public:
  void print_i() { cout << 2 << " inside D\n";}
};

main()
{
  B *pb = new B(); D *pd = new D(); B *p;
  pb -> print_i();  //should print 1 inside B
  pd -> print_i();  //should print 2 inside D
  pb = pd;
  pb -> print_i();  //should print 2 inside D
}
```