

## CSE 143

### Scope Review and Overloading

[Chapter 8, pp. 377-381]

11/3/00 T-1

## Review: Scope

- The "scope" of a name (identifier) is the part of the program where it is known
  - Mostly concerned with variables, parameters, and function names
- Function/block scope: starts where name is declared, ends at end of function or block
- Class scope: name can be used inside the class
  - including inside method implementations
  - `::` can be used to put you back inside a class scope

11/3/00 T-2

## Review: Function scope

- Function/block scope: starts where name is declared, ends at end of function or block

```
void Snork(int a)    //scope of a starts here
{
    ...
    int b;           //scope of b starts here
    ...
    while ( a > b ){
        int c = 0;    //scope of c starts here
        ...
    }                //scope of c ends here
    ...
}                    //scope of a and b end here
//the name "Snork" is still in scope!
```

11/3/00 T-3

## Reusing Identifiers

- Variable names may be repeated in different scopes
  - but often it's terrible style

```
int aVar; //global
void Snork (int aVar)
{
    ...
    while ( a > b ){
        int aVar = 0;
        ...
        aVar = aVar + 1; //which aVar?
    }
}
```

11/3/00 T-4

## Overloading

- Different functions in **same** scope can have same name if argument list is different
  - Function name is said to be **overloaded**
  - Applies to methods or non-methods
  - Constructors are common example of overloading
- In C++, operators (+, \*, =, [], etc.) can also be overloaded

11/3/00 T-5

## Overloading vs. Overriding

- **Overriding**: same function name and signature in derived class, overrides base class
  - only one of the two is in scope at a give time
  - "virtual" concept applies only to overriding
- **Overloading**: same function name and different signature
  - both functions are in scope at same time
  - Compiler determines which function to call at compile-time (statically)

11/3/00 T-6

## Resolving Overloaded Functions

To "resolve" mean to decide which version of the overloaded function is being called

- Determined by matching actual arguments against possible formal arguments
- Compiler gives error if not exactly one matches
- If match is not exact, automatic type conversions are used

constructors might be called, etc.

Complete matching algorithm rather complex

11/3/00 T-7

## Matching Algorithm

- Function declarations

```
void Snark(int);  
void Snark(double);  
void Snipe(char []);  
void Snipe(double);  
void Sneep(char);  
void Sneep(double);
```

```
Snark(1);           // Integer Snark  
Snipe(1);           // Double Snipe  
Sneep(1);           // Ambiguous
```

11/3/00 T-8

## Example of Resolving

- Function declarations

```
void PrintData(int data) {  
    cout << "int = " << data << endl; }  
  
void PrintData(char data) {  
    cout << "char = '" << data << "'\n"; }  
  
void PrintData(double data) {  
    cout << "double = " << data << endl; }
```

- Which calls are valid? Which version is called?

```
PrintData(3);  
PrintData("Hello");  
PrintData(3.14159);  
PrintData('m');
```

11/3/00 T-9

## Overloaded Operators

- For convenience, can define functions named +, -, \*, =, /, ==, etc. on classes
  - Gives natural expression to some operations
  - Very confusing if abused
  - Almost all C++ operators may be overloaded
- Operator functions may be members or "friends" if access to private data needed
  - At least one of the arguments must be a class!

11/3/00 T-10

## Example Prototype

- Function "names" are `operator+`, `operator-`, ...

```
IntList operator+(const IntList &s,  
                  const IntList &t);
```

- Several ways of setting them up

- The above example is probably a "friend" function

- Could also have a member function

```
IntList & IntList::operator+  
    (const IntList &rhs) const;
```

11/3/00 T-11

## Overloading <<

- Operator << can also be overloaded for new types
- Syntax is rather arcane
- Example:

```
// write Complex number as <real,imag>  
friend  
ostream& operator<<(ostream &s, Complex z) {  
    s << "representation of z";  
    return s;  
}
```

Only part of prototype  
that varies for new type

- Need to return reference to stream so chained I/O will work (cout << x << y << ...)

11/3/00 T-12