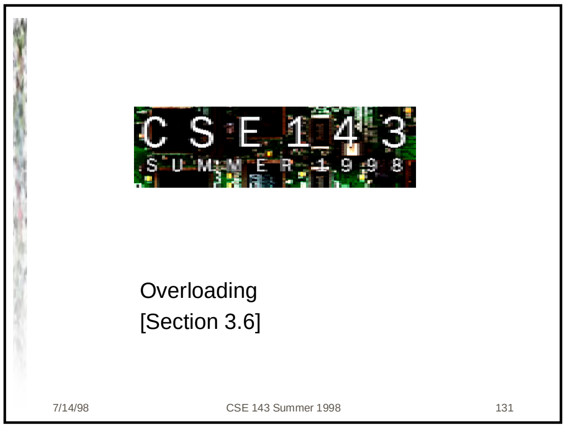
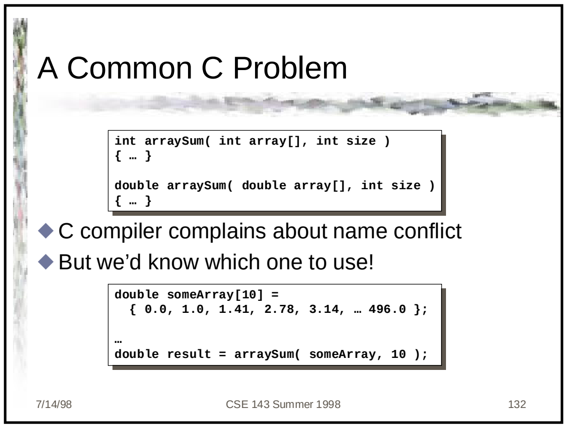


CSE 143 SUMMER 1998

Overloading
[Section 3.6]

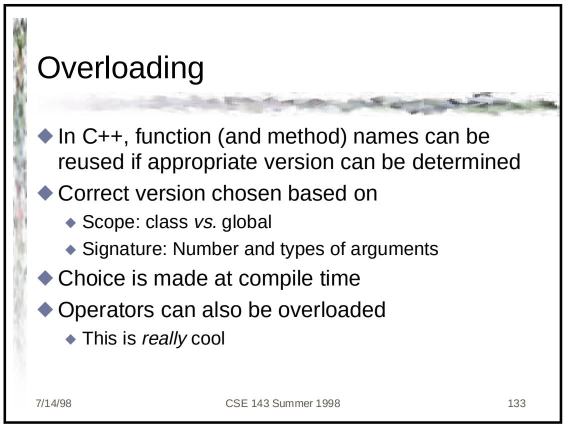


A Common C Problem

```
int arraySum( int array[], int size )
{ … }

double arraySum( double array[], int size )
{ … }
```

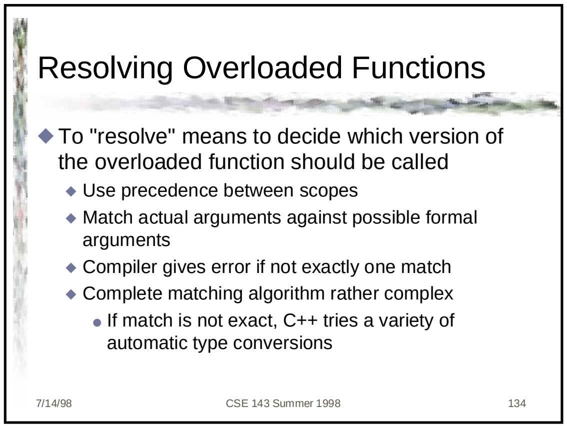
- C compiler complains about name conflict
- But we’d know which one to use!

```
double someArray[10] =
  { 0.0, 1.0, 1.41, 2.78, 3.14, … 496.0 };

…
double result = arraySum( someArray, 10 );
```



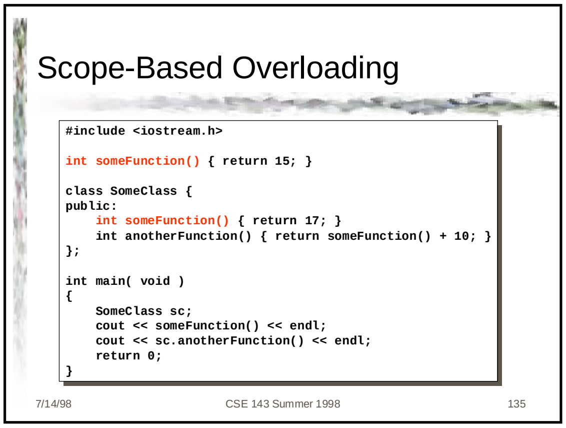
Overloading

- In C++, function (and method) names can be reused if appropriate version can be determined
- Correct version chosen based on
  - Scope: class *vs.* global
  - Signature: Number and types of arguments
- Choice is made at compile time
- Operators can also be overloaded
  - This is *really* cool



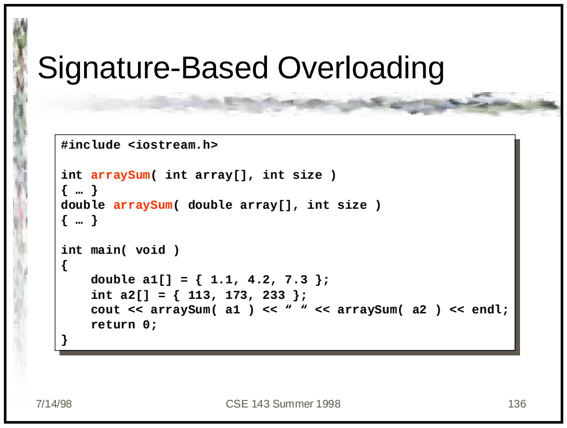
Resolving Overloaded Functions

- To "resolve" means to decide which version of the overloaded function should be called
  - Use precedence between scopes
  - Match actual arguments against possible formal arguments
  - Compiler gives error if not exactly one match
  - Complete matching algorithm rather complex
    - If match is not exact, C++ tries a variety of automatic type conversions



Scope-Based Overloading

```
#include <iostream.h>

int someFunction() { return 15; }

class SomeClass {
public:
    int someFunction() { return 17; }
    int anotherFunction() { return someFunction() + 10; }
};

int main( void )
{
    SomeClass sc;
    cout << someFunction() << endl;
    cout << sc.anotherFunction() << endl;
    return 0;
}
```



Signature-Based Overloading

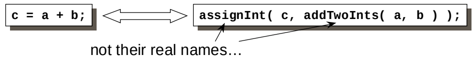
```
#include <iostream.h>

int arraySum( int array[], int size )
{ … }
double arraySum( double array[], int size )
{ … }

int main( void )
{
    double a1[] = { 1.1, 4.2, 7.3 };
    int a2[] = { 113, 173, 233 };
    cout << arraySum( a1 ) << “ “ << arraySum( a2 ) << endl;
    return 0;
}
```

Philosophy of Overloading

- Useful if one abstract operation is expressed by several similar functions
- Constructors are common example of overloading since all must have same name
- Advice: Avoid making excessive or spurious use of overloading!
  - Can make it difficult to read or understand programs

Operator Overloading

- Operators are just fancy syntax for function calls



```
c = a + b;
```

```
assignInt( c, addTwoInts( a, b ) );
```

not their real names…

- For convenience, can define functions named +, -, *, =, /, ==, etc. on your own classes
  - Gives natural expression to some operations
  - Very, *very* confusing if abused

Using Operator Overloading

- Operator functions are much like member functions
  - But have funny names
  - Several ways of calling them
- One very common example: << and >> operators for input and output
  - What are << and >> *really* for?

```
class ostream
{
public:
    ostream& operator <<( int n );
    ostream& operator <<( double d );
    …
};
```

Example: A Matrix Class (I)

```
class Matrix
{
public:
    Matrix();             // Create a zero matrix
    Matrix( double d );   // A multiple of identity

    bool operator ==( Matrix& m );    // Compare for equality

    Matrix operator +( Matrix& m );   // Add two matrices
    Matrix operator -( Matrix& m );   // Subtract
    Matrix operator *( Matrix& m );   // Multiply
    // Many other operations possible
private:
    …
};
```

matrix.h

Example: A Matrix Class (II)

```
#include “matrix.h”

bool Matrix::operator ==( Matrix& m )
{
    // Just like any other method in here
}

Matrix Matrix::operator *( Matrix& m )
{
    // And in here
}
```

matrix.cpp

```
#include “matrix.h”

int main( void )
{
    Matrix a, b, c;
    … c = a * b; …
}
```

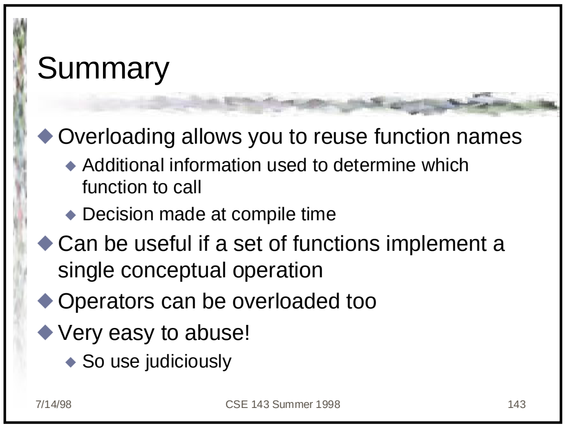
main.cpp

How Overloading Works

- Name mangling

```
int arraySum( int array[], int size );
double arraySum( double array[], int size );

… cout << “Hello”; …
```

```
int arraySum__FPii( int array[], int size );
double arraySum__FPdi( double array[], int size );

… __ls__7ostreamPCc( cout, “Hello” ); …
```



Summary

- Overloading allows you to reuse function names
  - Additional information used to determine which function to call
  - Decision made at compile time
- Can be useful if a set of functions implement a single conceptual operation
- Operators can be overloaded too
- Very easy to abuse!
  - So use judiciously

7/14/98 CSE 143 Summer 1998 143