

Function Overloading I

- we can overload functions on their arguments like so:

```
void
foo (int iarg)
{
    cout << "Inside foo with int arg: " << iarg << endl;
}
void
foo (double darg)
{
    cout << "Inside foo with double arg: " << darg << endl;
}
void
foo (string sarg)
{
    cout << "Inside foo with string arg: " << sarg << endl;
}
void
foo (int iarg, double darg, string sarg)
{
    cout << "Inside foo with lots of  args: "
        << iarg << ", " << darg << ", " << sarg << endl;
}
```

Function Overloading II

- you can't overload functions on their return values, why?
- recap: we had two definition of a `print()` function, one for the base class `MonteCarloSpecs` and another for the derived class `MHSpecs`
- here the definition of `MHSpecs::print()` is said to override the definition of `MonteCarloSpecs::print()`, its *not* called function overloading

Operator Overloading I

- the following built-in operators in C++ can be overloaded:
 - unary: `!`, `++`, `--`, `new`, `delete`, `new []`, `delete []`, `-->` etc.
 - binary: `!=`, `<<`, `>>`, `==`, `<`, `>`, `+=`, `-=`, `+`, `-`, `()`, `[]` etc.
- you cannot overload a define a new operator and overload it: e.g. `**` cannot be overloaded, you have write a function, say, `power` for it!
- operators that *can't* be overloaded: `.`, `.*`, `::`, `?:`

Operator Overloading II

- meaning of a overloaded operator @: (note there is no operator @, its just a “dummy” symbol, is you like!)
 - if the operator @ has been overloaded as a member function:
 - * unary: @obj means a function call `obj.operator@()`
 - * binary: `obj @ anotherObj` means a function call `obj.operator@(anotherObj)`
 - if the operator @ has been overloaded as a friend function:
 - * unary: @obj means a function call `operator@(obj)`
 - * binary: `obj @ anotherObj` means a function call `operator@(obj, anotherObj)`

Operator Overloading III

- consider the following declaration:

```
#include <iostream>
using std::ostream;
class MonteCarloSpecs {
    friend ostream & operator<< (ostream &ostream, MonteCarloSpecs const &mcs);
private:
    int n_iters;
    float time_in_secs;
    float prop_burn_in;
    double *log_density;
public:
    MonteCarloSpecs (int n_iters,
                    float time_in_secs,
                    float prop_burn_in = 0.05);
    ~MonteCarloSpecs (void);
    int get_n_iters (void) const;
    void set_n_iters (int n_iters);
    float get_time_in_secs (void) const;
    void set_time_in_secs (float time_in_secs);
    float get_prop_burn_in (void) const;
    void set_prop_burn_in (float prop_burn_in);
    void print (void) const;
    MonteCarloSpecs const & operator= (MonteCarloSpecs const &right);
};
```

Operator Overloading IV

- what is a friend function?
 - a friend function can access all the non-public members of a class although its not a member function (and hence doesn't need an object to be invoked upon)
- why consider a friend function at all?
 - for overloading operators << and >> you absolutely need to have a friend function because << and >> take an ostream object (e.g. cout) and istream (e.g. cin) as their first argument
 - hold your breath, lets move on here, we'll uncover the mystery in a bit!
 - in general, use friend functions when you absolutely have to (like the above cases), because they "violate" data hiding

Operator Overloading V

- implementation of a overloaded friend function:

```
ostream &
operator<< (ostream &ostream,
           MonteCarloSpecs const &mcs)
{
    ostream << "This Monte Carlo object:" << endl
              << "n_iters: " << mcs.get_n_iters( ) << endl
              << "time_in_secs: " << mcs.get_time_in_secs( ) << endl
              << "prop_burn_in: " << mcs.get_prop_burn_in( ) << endl
              << "log_density:" << endl;

    int ii;
    for (ii = 0; ii < mcs.get_n_iters( ) - 1; ++ii)
        ostream << mcs.log_density[ii] << ", ";
    ostream << mcs.log_density[ii] << endl;
    return ostream;
}
```

Operator Overloading VI

- when we use `cout << mcs1;` then we are essentially calling `operator<<(cout, mcs1);`
- similarly, `cout << mcs1 << mcs2` calls `operator<<(operator<<(cout, mcs1), mcs2);`
- thus the composition `cout << mcs1 << mcs2` is possible because the function `operator<<()` takes `cout` as the first argument and returns it back!
- thus `operator<<()` has to be a friend function not a member function

Operator Overloading VII

- implementation a overloaded member function:

```
MonteCarloSpecs const &
MonteCarloSpecs::operator= (MonteCarloSpecs const &right)
{
    set_n_iters(right.get_n_iters( ));
    set_time_in_secs(right.get_time_in_secs( ));
    set_prop_burn_in(right.get_prop_burn_in( ));
    for (int ii = 0; ii < get_n_iters( ); ++ii)
        log_density[ii] = right.log_density[ii];
    return *this;
}
```

Operator Overloading VIII

- question is: why can't we declare (and define) the previous function the following way?

```
void operator= (MonteCarloSpecs const &right);
```

- to answer it answer the following: which of the two declarations (and hence definitions) would support the following code snippet?

```
MonteCarloSpecs mcs1(10, 10, 0.1);  
MonteCarloSpecs mcs2(5, 5);  
MonteCarloSpecs mcs3(30, 30, 0.3);
```

```
mcs3 = mcs2 = mcs1;
```

- note `mcs3 = mcs2 = mcs1;` means `mcs3.operator=(mcs2.operator=(mcs1))`, does that help?

Operator Overloading IX

- some quick points, say, we have a class called `Foo` and a variable `Foo foo()`; was created with a constructor with no arguments(, assuming there is one such)
- the preincrement operator: `++foo`; means `foo.operator++()`, so the following has to be declared and defined

```
Foo & operator++ (void);
```
- the postincrement operator: `foo++`; means `foo.operator++(0)`, so the following has to be declared and defined

```
Foo operator++ (int dummy);
```
- note the `int dummy` is a flag to help the compiler distinguish between the preincrement and the postincrement operator
- the preincrement operator returns `Foo &` it needs returns a reference to `*this` which is the “incremented” version
- the postincrement operator returns `Foo` it needs returns a copy of the `*this` which is not “incremented yet”

Operator Overloading X

- the following type of overloading could be very useful:

```
class PoorlyImplementedNamedList {
private:
    int n_items;
    double *prices;
    string *names;

public:
    // lots of functions
    double operator[] (string const name);
    // lots of functions
};
```

- it could be used as:

```
PoorlyImplementedNamedList pinl;
// initialize the list
cout << pinl["item1Name"] << pinl["item2Name"] << endl;
```

Operator Overloading XI

- the following type of overloading could be very useful:

```
class PoorlyImplementedMatrix {
private:
    int n_rows, n_cols;
    double **vals;

public:
    // lots of functions
    double operator() (int row, int col);
    // lots of functions
};
```

- it could be used as:

```
PoorlyImplementedMatrix mm;
// initialize the matrix
cout << mm(0, 0) << mm(1, 0) << endl;
```

Code Files

```
prog8.H  
prog8.C  
prog8Makefile
```