Welcome
Notes
Welcome to OO
Object Terminology
Modular Programming
Types Overloading
Dynamic Memory
Member Functions
Construction
Current Object
Member Operators
Class + Resources
Helper Functions
Input Output
Derived Classes
Derived Functions
Virtual Functions
Abstract Classes
Templates
Polymorphism
I/O Refinements
D C + Resources
Standards
Bibliography
Library Functions
ASCII Sequence
Operator Precedence
C++ and C
Workshops Workshops
Assignments
Handouts

## Part C - Encapsulation

# Helper Functions

Support a class definition with global function definitions
Describe the syntax for overloading operators that support a class
Grant a global function access to the private members of a class

*"Avoid membership fees: Where possible, prefer making functions nonmember non-friends"*
*(Sutter, Alexandrescu, 2005)*

Free | Operators | Friends | Summary | Exercise

In object-oriented programming, not all instructions that support a class need to be included in the class definition.  A well-encapsulated class can accept external support in the form of global functions containing additional logic.  We call these functions *helper functions*.  Helper functions access the objects of a class solely through their parameters, all of which are explicit.  A typical helper function includes at least one parameter of the class type that it supports.

This chapter describes the definition of helper functions, including helper operators, and discusses the granting of privileged access to the private members of a class.

## FREE HELPERS

A free or loosely coupled helper function is a function that obtains all of its information from the public member functions of the class that it supports.  A free helper function does not require access to the private members of its class.  The coupling between a free helper function and its class is minimal, which is an ideal design solution.

### Comparison

Consider a helper function that compares two objects of the same class.  The function returns `true` if the objects have the same data values and `false` if they differ.

#### Example

Let us add three queries (`getStudentNo()`, `getNoGrades()` and `getGrade()`) to our `Student` class definition and a helper function named `areIdentical()` as support.  For conciseness, let us assume that all grades are stored in static memory.  We insert the prototype for our helper function into the header file *after* the class definition:

```
// Student.h

const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    void display() const;
    int getStudentNo() const { return no; }
    int getNoGrades() const { return ng; }
    float getGrade(int i) const { return i < ng ? grade[i] : 0.0f; }
};

bool areIdentical(const Student&, const Student&);
```

The implementation file contains the definition of our helper function.

```
// Student.cpp

#include <iostream>
using namespace std;
#include "Student.h"

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int n) {
    *this = Student(n, nullptr, 0);
}

Student::Student(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        *this = Student();
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

bool areIdentical(const Student& lhs, const Student& rhs) {
    bool same = lhs.getStudentNo() == rhs.getStudentNo() &&
     lhs.getNoGrades() == rhs.getNoGrades();
    for (int i = 0; i < lhs.getNoGrades() && same; i++)
```

```
         same = lhs.getGrade(i) == rhs.getGrade(i);
     return same;
 }
```

The following client code compares the two objects:

```
 // Compare Objects
 // compare.cpp

 #include <iostream>
 #include "Student.h"
 using namespace std;

 int main () {
     float gh[] = {89.4f, 67.8f, 45.5f};
     Student harry(1234, gh, 3), harry_(1234, gh, 3);
     if (areIdentical(harry, harry_))
         cout << "are identical" << endl;
     else
         cout << "are different" << endl;
 }
```

```
are identical
```

### The Cost of Upgrading Freedom

Free helper functions use public queries to access information that is otherwise inaccessible. If we add a data member to the class, we may also need to add a query to access its value. As we add data members, the class definition grows with new queries. We call this growth *class bloat*.

One alternative to class bloat that admits upgradability is friendship (see below).

## HELPER OPERATORS

Helper operators are global functions that overload operators. Candidates for helper operators are operators that do not change the values of their operands as shown in the table below.

| Effect on Operand(s) | Candidate | Operands | Operator |
|---|---|---|---|
| Left Operand Changes | Member | 0 | `++ -- - + ! & *` |
| | | 1 | `= += -= *= /= %=` |
| Neither Operand Changes | Helper | 2 | `+ - * / % == != >= <= > < << >>` |

### Identity Comparison

To improve readability, let us replace the **`areIdentical()`** function defined above with an overloaded **`==`** operator that takes two **`Student`** operands. The header file for the **`Student`** class now contains:

```
 // Student.h

 const int NG = 20;

 class Student {
     int no;
     float grade[NG];
     int ng;
 public:
     Student();
     Student(int);
     Student(int, const float*, int);
     void display() const;
     int getStudentNo() const { return no; }
     int getNoGrades() const { return ng; }
     float getGrade(int i) const { return i < ng ? grade[i] : 0.0f; }
 };

 bool operator==(const Student&, const Student&);
```

The implementation file contains defines this helper operator:

```
 bool operator==(const Student& lhs, const Student& rhs) {
     bool same = lhs.getStudentNo() == rhs.getStudentNo() &&
      lhs.getNoGrades() == rhs.getNoGrades();
     for (int i = 0; i < lhs.getNoGrades() && same; i++)
         same = lhs.getGrade(i) == rhs.getGrade(i);
     return same;
 }
```

The following client code compares the two objects:

```
 // Compare Objects
 // compare.cpp

 #include <iostream>
 #include "Student.h"
 using namespace std;

 int main () {
     float gh[] = {89.4f, 67.8f, 45.5f};
     Student harry(1234, gh, 3), harry_(1234, gh, 3);
     if (harry == harry_)
         cout << "are identical" << endl;
     else
         cout << "are different" << endl;
 }
```

```
are identical
```

## Addition

Consider an expression that adds a single grade to a `Student` object and evaluates to a copy of the updated object.  To implement this operation, let us overload the `+` operator for a `Student` object as the left operand and a `float` as the right operand.

As part of the class definition, we include the `+=` operator described in the preceding chapter on Member Operators.  The header file for the `Student` class now contains:

```
 // Student.h

 const int NG = 20;

 class Student {
     int no;
     float grade[NG];
     int ng;
 public:
     Student();
     Student(int);
     Student(int, const float*, int);
     const Student& operator+=(float);
     void display() const;
     int getStudentNo() const { return no; }
     int getNoGrades() const { return ng; }
     float getGrade(int i) const { return i < ng ? grade[i] : 0.0f; }
 };

 bool operator==(const Student&, const Student&);
 Student operator+(const Student&, float);
```

We maintain loose coupling by initializing a new `Student` object to the left operand and calling the `+=` member operator on that object to add the single grade:

```
 Student operator+(const Student& s, float grade) {
     Student copy = s; // makes a copy
     copy += grade;    // calls the += operator on copy
     return copy;      // return updated copy
 }
```

For symmetry, we overload the addition operator for identical operand types but in reverse order. The complete header file contains:

```
 // Student.h

 const int NG = 20;

 class Student {
     int no;
     float grade[NG];
     int ng;
 public:
     Student();
     Student(int);
     Student(int, const float*, int);
     const Student& operator+=(float);
     void display() const;
     int getStudentNo() const { return no; }
     int getNoGrades() const { return ng; }
     float getGrade(int i) const { return i < ng ? grade[i] : 0.0f; }
 };

 bool operator==(const Student&, const Student&);
 Student operator+(const Student&, float);
 Student operator+(float, const Student&);
```

Our implementation calls the first version with the arguments reversed:

```
 // Student.cpp

 #include <iostream>
 using namespace std;
 #include "Student.h"

 Student::Student() {
     no = 0;
     ng = 0;
 }

 Student::Student(int n) {
     float g[] = {0.0f};
     *this = Student(n, g, 0);
 }

 Student::Student(int sn, const float* g, int ng_) {
     bool valid = sn > 0 && g != nullptr && ng_ >= 0;
```

```
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        *this = Student();
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

const Student& Student::operator+=(float g) {
    if (no != 0 && ng < NG && g >= 0.f && g <= 100.f)
        grade[ng++] = g;
    return *this;
}

bool operator==(const Student& lhs, const Student& rhs) {
    bool same = lhs.getStudentNo() == rhs.getStudentNo() &&
     lhs.getNoGrades() == rhs.getNoGrades();
    for (int i = 0; i < lhs.getNoGrades() && same; i++)
        same = lhs.getGrade(i) == rhs.getGrade(i);
    return same;
}

Student operator+(const Student& student, float grade) {
    Student copy = student; // makes a copy
    copy += grade;          // calls the += operator on copy
    return copy;            // return updated copy
}

Student operator+(float grade, const Student& student) {
    return student + grade; // calls operator+(const
                            //    Student&, float)
}
```

The following client code produces the results shown on the right:

```
// Helper Operator
// helper-addition-operator.cpp

#include <iostream>
#include "Student.h"
using namespace std;
```

```
int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};                              1234:
    Student harry(1234, gh, 3);                                        89.40
    harry = harry + 63.7f;                                             67.80
    harry.display();                                                   45.50
}                                                                      63.70
```

## FRIENDSHIP

Friendship grants helper functions access to the private members of a class. By granting friendship status, a class lets a helper function access to any of its private members: data members or member functions. Friendship minimizes class bloat.

To grant a helper function friendship status, we declare the function a *friend* and place its declaration inside the class definition. A friendship declaration takes the form

```
friend Type identifier(type [, type, ...]);
```

where ***Type*** is the return type of the function and ***identifier*** is the function's name.

For example:

```
// Student.h

const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    const Student& operator+=(float);
    void display() const;
    friend bool operator==(const Student&, const Student&);
};

Student operator+(const Student&, float);
Student operator+(float, const Student&);
```

Our implementation looks like:

```
// Student.cpp

#include <iostream>
using namespace std;
#include "Student.h"

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int n) {
    float g[] = {0.0f};
    *this = Student(n, g, 0);
}

Student::Student(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        *this = Student();
    }
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

const Student& Student::operator+=(float g) {
    if (no != 0 && ng < NG && g >= 0.f && g <= 100.f)
        grade[ng++] = g;
    return *this;
}

Student operator+(const Student& student, float grade) {
    Student copy = student; // makes a copy
    copy += grade;          // calls the += operator on copy
    return copy;            // return updated copy
}

Student operator+(float grade, const Student& student) {
    return student + grade; // calls operator+(const
                            //    Student&, float)
}

bool operator==(const Student& lhs, const Student& rhs) {
    bool same = lhs.no == rhs.no &&  lhs.ng == rhs.ng;
    for (int i = 0; i < lhs.ng && same; i++)
        same = lhs.grade[i] == rhs.grade[i];
    return same;
}
```

We have added the keyword **`friend`** to the declaration within the class definition.  We do not apply the keyword to the function definition.

The following client code compares the two objects:

```
// Friends
// friends.cpp

#include <iostream>
#include "Student.h"
```

```
using namespace std;

int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry(1234, gh, 3), harry_(1234, gh, 3);
    harry_ += 63.7f;
    if (harry == harry_)
        cout << "are identical" << endl;
    else
        cout << "are different" << endl;                        are different
}
```

## The Cost of Friendship

A class definition that grants friendship to a helper function allows that function to alter the values of its private data members. Granting friendship pierces encapsulation.

As a rule, we grant friendship judiciously only to helper functions that require both *read and write* access to the private data members. Where read-only access is all that a helper function needs, using queries is probably more advisable.

Friendship is the strongest relationship that a class can grant an external entity.

## Friendly Classes (Optional)(Optional)

One class can grant another class access to its private members. A class friendship declaration takes the form

```
friend class Identifier;
```

where ***`Identifier`*** is the name of the class to which the host class grants friendship privileges.

For example, an **`Administrator`** class needs access to all information held within each **`Student`** object. To grant this access, we simply include a class friendship declaration within the **`Student`** class definition

```
// Student.h

const int M = 13;

class Student {
    int no;
    char grade[M+1];
public:
    Student();
    Student(int);
    Student(int, const char*);
    void display() const;
    const Student& operator+=(char);
    friend bool areIdentical(const Student&, const Student&);
    friend class Administrator;
};
```

## No Reciprocity, Transitivity or Exclusivity (Optional)(Optional)

Friendship is neither reciprocal, transitive nor exclusive. Just because one class is a friend of another class does not mean that the latter is a friend of the former. Just because a class is a friend of another class and that other class is a friend of yet another class does not mean that the latter class is a friend of either of them. A friend of one class may be a friend of any other class.

Consider three classes: a **`Student`**, an **`Administrator`** and an **`Auditor`**.

- Let the **`Auditor`** be a friend of the **`Administrator`** and the **`Administrator`** be a friend of the **`Student`**
- Just because the **`Auditor`** is a friend of any **`Administrator`** and the **`Administrator`** is a friend of any **`Student`**, the **`Administrator`** is not necessarily a friend of the **`Auditor`** and the **`Student`** is not necessarily a friend of the **`Administrator`** (lack of reciprocity)

- Just because the **`Auditor`** is a friend of any **`Administrator`** and the **`Administrator`** is a friend of any **`Student`**, the **`Auditor`** is not necessarily a friend of any **`Student`** (lack of transitivity)

## SUMMARY

- a helper function is a global function that supports a class
- a helper function refers to the class that it supports through its explicit parameter(s)
- a helper operator is typically an operator that does not change the value of its operands
- a friend function has direct access to the private members of the class that granted the function friendship
- friendship is neither reciprocal, transitive, nor exclusive
- free helper functions reduce coupling at the cost of bloating a class
- friendly helper functions reduce bloating at the cost of piercing encapsulation

## EXERCISES

- Complete the Handout on Helper Operators.
- Complete the Workshop on Expressions

print this page Top
Previous: Classes and Resources Next: Input and Output Operators

ICT ICT Home Outline Outline Timeline
Notes IPC Notes IPC Notes MySeneca MySeneca Workshops Workshops Assignments Instructor