There are two ways to pass arguments to functions: pass-by-value and pass-by-reference.

- **pass-by-value**
  - A copy of the argument’s value is made and passed to the called function.
  - Changes to the copy do not affect the original variable’s value in the caller.
Reference Parameters

• **pass-by-reference**
  
  – The caller passes the address of its data.
  
  – The caller gives the called function the ability to access the caller’s data directly and to modify it if the called function chooses to do so.
Reference Parameters

- A reference parameter is an alias for its corresponding argument in the function call.
- Use the ampersand (&) to indicate that the parameter is passed by reference.
- All operations performed on the alias (i.e. the reference) are actually performed on the original variable. The alias is simply another name for the original variable.
### Reference Parameters

**"pass-by-value in C" example:**
```c
#include <stdio.h>

void swap(int first, int second); // prototype

int main()
{
    int x = 5;
    int y = 6;
    printf("Before: x = %d  y = %d\n", x, y);
    swap(x, y);
    // were the integers swapped?
    printf("After:  x = %d  y = %d\n", x, y);
    return 0;
}

void swap(int first, int second)
{
    int temp;
    temp = first;
    first = second;
    second = temp;
}

Output:
Before: x = 5  y = 6
After:  x = 5  y = 6
```

**"simulating pass-by-reference in C" example:**
```c
#include <stdio.h>

void swap(int *first, int *second); // prototype

int main()
{
    int x = 5;
    int y = 6;
    printf("Before: x = %d  y = %d\n", x, y);
    swap(&x, &y);
    // were the integers swapped?
    printf("After:  x = %d  y = %d\n", x, y);
    return 0;
}

void swap(int *first, int *second)
{
    int temp;
    temp = *first;
    *first = *second;
    *second = temp;
}

Output:
Before: x = 5  y = 6
After:  x = 6  y = 5
```
Reference Parameters in C++

- C++ has true reference "pass-by-reference".
- Use the & operator to denote a reference parameter, e.g.
  ```
  void swap(int &a, int &b);
  ```
  or
  ```
  void swap(int &, int &);
  ```
"pass-by-reference in C++" example:

```cpp
#include <iostream>
using namespace std;

void swap(int &first, int &second); // prototype

int main()
{
    int x = 5;
    int y = 6;
    cout << "Before: x = " << x << " y = " << y;
    swap(x, y);
    //were the integers swapped?
    cout << "After:  x = " << x << " y = " << y
         << endl;
    return 0;
}

void swap(int &first, int &second) {
    int temp;
    temp = first;
    first = second;
    second = temp;
}
```

Output:
Before: x = 5  y = 6
After: x = 6  y = 5
Reference Parameters

- Pass-by-reference is easier and more natural for some programmers than simulating pass-by-reference with pointers.
- Some programmers prefer using pointers, because * in the parameter list and the & in the call, make it explicitly clear that the called function will alter the caller’s data.
- With call-by-reference, the parameter list in the function call is identical to the parameter list for call-by-value.
Function Overloading

• A class may have several methods that have the same name. This is referred to as function overloading.

• When a function name is overloaded, the implementation that is actually used is the one whose formal parameters match the actual arguments being passed by the caller in both number of arguments and type of argument. Therefore, it is mandatory that each implementation has distinguishable parameters.
Operator Overloading (Restrictions)

- Most operators can be overloaded. The ones that cannot are
  - . (member access),
  - .* (member access through pointer to member),
  - :: (scope resolution),
  - ?: (ternary conditional), and
  - sizeof
Operator Overloading (Restrictions)

- New operators such as **, <>, or &| cannot be created.

- The overloads of operators &&, ||, and , (comma) lose their special properties: short-circuit evaluation and sequencing.

- The overload of operator -> must either return a raw pointer or return an object (by reference or by value), for which operator -> is in turn overloaded.

- It is not possible to change the precedence, grouping, or number of operands of operators.
Operator Overloading

- Operator overloading is actually part of the function overloading mechanism. The name of the function for an overloaded operator is `operator` followed by the operator symbol, e.g.,

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>operator+</td>
</tr>
<tr>
<td>-</td>
<td>operator-</td>
</tr>
<tr>
<td>*</td>
<td>operator*</td>
</tr>
<tr>
<td>&lt;=</td>
<td>operator&lt;=</td>
</tr>
<tr>
<td>-&gt;</td>
<td>operator-&gt;</td>
</tr>
</tbody>
</table>
Operator Overloading

• Operator overloading is restricted to existing operators. Thus it is *not legal* to try to overload `**` operator `**`

• Operators can be overloaded as
  – instance member functions, or as
  – friend functions
Operator Overloading

• The operator functions work "almost" exactly like "regular" functions. They can even be invoked using their operator+ or operator- names. Keeping this fact in mind will remove much of the mystery from how they work and how they must be implemented.

• The "almost" qualifier above reflects necessity to remember that almost all C operators take either one or two operands. Thus an operator function has at most two parameters.
Operator Overloading

- The C addition operator takes two operands: \( a + b \)
- Therefore the \textit{operator}+ function will have two parameters: the first will represent the \textit{left side operand} and the second the \textit{right side} operand.
- The C \textbf{logical not} operator takes one operand: \texttt{!value}
- Therefore the \textit{operator}! function will have one parameter and it will represent the \textit{right side} operand.
Operator Overloading

- Operator overloading must preserve the "aryness" (unary or binary) nature of the operator. Thus, the ! operator could be overloaded to compute the length of a single vector but could not be used to compute a dot product. The operators &, *, +, - have both binary and unary versions that may be overloaded separately.

- It is not possible to change the precedence or associativity of an overloaded operator.
Operator Overloading

- Most operator functions return a value that replaces the operator and its operand(s) in an expression. However, that is not mandatory. For example a side effect operator such as ++ may not need to return a value.
Overloading Operators as Instance Members

A binary operator that is overloaded as an instance member needs only one parameter, which represents the operand on the right:

class OpClass
{
    private:
        int x;
    public:
        OpClass operator+(OpClass right);
};
Overloading Operators as Instance Members

- The left operand of the overloaded binary operator is the calling object
- The implicit left parameter is accessed through the `this` pointer

```cpp
OpClass OpClass::operator+(OpClass r) {
    OpClass sum;
    sum.x = this->x + r.x;
    return sum;
}
```
Invoking an Overloaded Operator

- Operator can be invoked as a member function:
  
  ```cpp
  OpClass a, b, s;
  s = a.operator+(b);
  ```

- It can also be invoked in the more conventional manner:
  
  ```cpp
  OpClass a, b, s;
  s = a + b;
  ```
Overloading Assignment

- Overloading the assignment operator solves problems with object assignment when an object contains pointer to dynamic memory.
- Assignment operator is most naturally overloaded as an instance member function
- It needs to return a value of the assigned object to allow cascaded assignments such as
  
  \[ a = b = c; \]
Assignment overloaded as a member function:

class CpClass
{
    int *p;
public:
    CpClass(int v=0)
    {
        p = new int; *p = v;
    }
    ~CpClass()
    {
        delete p;
    }
    CpClass &operator=(CpClass);  
};
Overloading Assignment

Implementation returns a value:

```cpp
CpClass CpClass::operator=(CpClass r) {
    *p = *r.p;
    return *this;
};
```

Invoking the assignment operator:

```cpp
CpClass a, x(45);
a.operator=(x); // either of these
a = x;          // lines can be used
```
Overloading `<<` and `>>`

- The overloaded operators `<<` and `>>` can be used to print and to read numeric and character string values to stdout and stderr and from stdin, respectively.

- True to form, they can be further overloaded to print and read a complete object, such as a Rational object.
Overloading << and >>

Given the following class declaration for class Complex:

```cpp
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex{
public:
    Complex( );
    Complex(double realInit, double imaginaryInit);
    void setComplex(double realInit, double imaginaryInit);
    Complex add(const Complex &rhs) const;
    Complex subtract(const Complex &rhs) const;
    double magnitude( );
    void printComplex( );

private:
    double real;
    double imaginary;
};
#endif
```
Operator Overloading

We can overload the following operators to add, subtract, multiply, and two objects:

Complex operator+ (const Complex &rhs) const;
Rational operator- (const Complex &rhs) const;
Rational operator* (const Complex &rhs) const;
Overloading +

Complex::operator+ (const Complex &rhs) const
{
    Complex sum;
    sum.real = real + rhs.real;
    sum.imaginary = imaginary + rhs.imaginary;
    return sum;
}
Overloading \(<<\) and \(>>\)

Given: `Complex c;`

We want to be able to do something like

```cpp
cout << c;  or  cin >> c;
```

where `r1` is a `Rational` object.

- Since the left hand side of `>>` or `<<` is not a `Complex`, we must use the friend function form. So in `complex.h` we include:

```cpp
friend ostream &operator<<(ostream &out, const Complex &c);
```
Overloading << and >>

We provide the implementation in either rational.cpp or inline in the class definition.

```cpp
ostream &operator<<(ostream &out, const Complex &c) {
    char op = '+';
    if (imaginary < 0)
        op = '-';
    out << "(" << real
        << " " << op
        << " " << fabs(imaginary) << "i")";

    return(out);
}
```
Overloading \texttt{<<} and \texttt{>>}

- Note that our new overload just uses the built in overload of \texttt{<<} to output each component of the Complex object.
- This is the key to cascading application of the \texttt{<<} operator.
Overloading `<<` and `>>>`

The implementation of `>>>` is similar

```cpp
istream &operator>>>(istream &in, const Complex &c) {
    in >> c.real >> c.imaginary;
    return in;
}
```
Overloading Types of Operators

- `++`, `--` operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a `bool` value
- Overloaded stream operators `>>, <<` must return `istream, ostream` objects and take `istream, ostream` objects as parameters
Overloaded [] Operator

• Can be used to create classes that behave like arrays, providing bounds-checking on subscripts

• Overloaded [] returns a reference to object, not an object itself
Conversion Operators are member functions that tell the compiler how to convert an object of the class type to a value of another type.

The conversion information provided by the conversion operators is automatically used by the compiler in assignments, initializations, and parameter passing.
Syntax of Conversion Operators

• Conversion operator must be a member function of the class you are converting from

• The name of the operator is the name of the type you are converting to

• The operator does not specify a return type
Conversion Operator Example

• To convert from a class \texttt{IntVal} to an integer:

```cpp
class IntVal
{
    public:
        IntVal(int a = 0)
        {
            x = a;
        }
        operator int(){
        { return x;
        }
    private:
        int x;
};
```
Conversion Operator Example

• Automatic conversion during assignment:

```cpp
IntVal obj(15);
int i;
i = obj;
cout << i; // prints 15
```
Convert Constructors

Convert constructors are constructors that take a single parameter of a type other than the class in which they are defined.

class CCClass
{
    int x;

    public:
        CCClass() //default
        CCClass(int a, int b);
        CCClass(int a);    //convert
        CCClass(string s); //convert
};
Example of a Convert Constructor

The C++ `string` class has a convert constructor that converts from C-strings:

```cpp
class string
{
public:
    string(char *);  //convert
    ...
};
```
Uses of Convert Constructors

• They are automatically invoked by the compiler to create an object from the value passed as parameter:

  ```
  string s("hello");  //convert C-string
  CCClass obj(24);    //convert int
  ```

• The compiler allows convert constructors to be invoked with assignment-like notation:

  ```
  string s = "hello"; //convert C-string
  CCClass obj = 24;   //convert int
  ```
Uses of Convert Constructors

- Convert constructors allow functions that take the class type as parameter to take parameters of other types:

```cpp
void myFun(string s); // needs string
myFun("hello"); // accepts C-string

void myFun(CCClass c);
myFun(34); // accepts int
```