Overloading

Function overloading

- Call to a function should be based on context
  - Which is better?
    - ∗ `student_print ( student )`, or `student.print();`
    - ∗ `date_print ( date )`, or `date.print()`
- Allows for the use of same name for multiple functions, for example, `length()` to compute the length of a string, list, and vector
- Compiler calls the appropriate function depending on the parameters

```
int length ( char * s )       // Length of a string
{
    char * t = s;
    while ( *t++ );
    return ( t - s );
}

double length ( double *vec, int n )   // Length of a vector in n-space
{
    for ( double r = 0; n--; r += vec[n] * vec[n] );
    return ( sqrt ( r ) );
}

int n = length ( "Harry" );
double l, x[3] = { 1, 0, -2 };
l = length ( x, 3 );
```

  - The overloaded methods can belong to the same scope in which case they are differentiated by parameter types
    - ∗ Compiler cannot generate unique internal identifiers if it uses only the scope of the function names
    - ∗ Compiler must *mangle* the names of the parameter types with the function name
    - ∗ The above global function `length()` can produce internal names that look like `_length_charp` and `_length_vecp_int`
    - ∗ Mangling varies from compiler to compiler and hence, you may not be able to use functions generated by one compiler in another
- The constructor methods are by necessity overloaded
  - We have looked at default constructors and parameterized constructors
- The compiler adds type conversions (such as `int` to `double`) if necessary to make the call conform to the arguments in the function
- The following steps are used to find a matching function
  1. If an exact match of the argument type is found, use it
  2. If there is a unique function that matches after the following *promotions*, use that function

`char`	→	`int`
`unsigned char`	→	`int`
`short`	→	`int`
`unsigned short`	→	`(sizeof(short) < sizeof(int)) ? int : unsigned`
`float`	→	`double`

3. If there is a unique function that matches after other standard type conversions, use that function
4. If there is a unique user-defined conversion achieving a match, use that function

- Matches
  - `0` is an exact match for an `int` and can be converted to a pointer or `double` by a standard conversion
  - `char` and `short` are not considered exact matches for `int`
  - Standard conversions that might lead to information loss (such as `int` to `char`, `double` to `int`) are considered for matching
  - Use casts `char( ch )`, `int( x )` if necessary
- One can also think of member functions as being overloaded as well
  - Let us consider member functions `list::length()`, `queue::length()`, and `vec3::length()`
  - In compiling `x.length()`, one of the above three is selected depending on the class to which `x` belongs, and no type conversion is applied

Operator overloading

- The built in operators can be defined to act on structured data types by defining special functions

```
string operator+ ( const string&, const string& );
```

- If the above function is defined, the concatenation of two strings can be achieved as follows

```
string a;
string b ( "Harry" ), c ("Hacker");
a = b + c;      // Concatenate strings
```

- The following operators can be overloaded:

```
+   -   *   /   %   ^   &   |   ~   !   ,   =   <   >   <<  >>  <=  >=  ==  &&
||  ++  --  +=  -=  *=  /=  %=  ^=  &=  |=  !=  <<= >>= ->  ()  new delete
```

- The `operator` function can be attached only to existing operators
- You cannot design new operators, such as `|x|` for absolute values
- You cannot change the precedence, prefix/postfix application, or the arity
- `operator->` takes no argument and must return a pointer to a structure
- Operator functions must take at least one `struct` (or `class`) argument; hence the following is wrong

```
char* operator+ ( char*, char* );
```

- Operator functions can be either global functions or members of a class
  - `operator=`, `operator[]`, `operator()`, and `operator->` are exceptions to the above rule, and must be member functions only
- No special meaning is assigned to any of the operators
  - It is possible to define `operator+` to denote vector subtraction
  - This also rhymes with the fact the operators for `cin >> x` and `cout << x` are overloaded from the shift operators >> and <<

- The operators +, -, *, and & can be overloaded as unary or binary operators
- Minor complication with the ++ and -- operators
  - Both have prefix and postfix versions
  - Both the operators are unary, and hence, the number of arguments can not be used to distinguish between them

```
class complex
{
    double re, im;

    public:
        complex ( double r = 0, double i = 0 )    // Constructor
        {
            re = r;
            im = i;
        }
        double real() { return re; }
        double imag() { return im; }
        complex inv();                 // Inverse
        complex operator- ()           // Negative
        {
            return ( complex ( -re, -im ) );
        }
        friend complex operator+ ( const complex&, const complex & );
        friend complex operator- ( const complex&, const complex & );
        friend complex operator* ( const complex&, const complex & );
        friend complex operator/ ( const complex&, const complex & );
        complex& operator+= ( const complex& );
        complex& operator-= ( const complex& );
        complex& operator*= ( const complex& y )
        {
            return ( *this = *this * y );
        }
        complex& operator/= ( const complex& y )
        {
            return ( *this = *this * y.inv() );
        }
        friend int operator== ( const complex&, const complex& );
        friend int operator!= ( const complex&, const complex& );
};

complex complex::inv()
{
    double norm = re * re + im * im;
    if ( ! norm )                     // (0, 0) has no inverse
        return ( *this );
    return ( complex ( re/ norm, -im/norm ) );
}

complex operator+ ( const complex& x, const complex& y )
{
    return ( complex ( x.re + y.re, x.im + y.im ) );
}

complex operator- ( const complex& x, const complex& y )
{
    return ( x + ( -y ) );
}

complex operator* ( const complex& x, const complex& y )
{
    return ( complex ( x.re * y.re - x.im * y.im , x.re * y.im + x.im * y.re ) );
}

complex operator/ ( const complex& x, const complex& y )
{
    return ( x * y.inv() );
}

complex& complex::operator+= ( const complex& y )
{
    re += y.re;
    im += y.im;
    return ( *this );
}
```

- Use reference variables for efficiency considerations
  - Pass only the address of the complex number rather than two doubles
- Most operators can be coded as friend functions

Overloaded operator[]

- Let us again look at the safe integer array class

```
class int_array
{
    int value[MAXSIZE];
    int lower, upper;          // Bounds of the array

    public:
        int_array ( int lo, int hi );    // Constructor function
        int& operator[] ( int );         // Accessing an element
};

int& int_array::operator[] ( int n )
{
    if ( n < lower || n > upper )
        error ( "Index out of bounds" );
    return ( value[n-lower] );
}
```

  - Because the function return type is int&, a[n] actually returns a pointer to a.value[n] and can be used on the left side of an assignment

- Associative array
  - A data structure that associates certain keys, typically strings, with other values
  - Example

    ```
    assoc_array a;
    a["Harry"] = 5.3;
    cout << a["Harry"];
    ```

  - A typical implementation consists of an array of strings and a parallel array of `double`
  - We can use a simple hashing scheme for the strings, resolving collisions through the next available entry in the string array

    ```
    class assoc_array
    {
        char * key[MAXENTRY];
        double val[MAXENTRY];
        char   buffer[BUFSIZE];  // Stores actual strings
        int    buf_end;

        int locate ( const char* );  // Find hash location of a string

        public:
            double& operator[] ( char * );
    };

    int assoc_array::locate ( const char * s )
    {
        int h = hash ( s ) % MAXENTRY;    // Compute key transformation
        int i = h;

        do
        {
            if ( ! ( key[i] && strcmp ( s, key[i] ) ) )
                return ( i );
            if ( ++i >= MAXENTRY )
                i = 0;
        }
        while ( i != h );

        return ( -1 );                   // Not found
    }

    double& assoc_array::operator[] ( char * s )
    {
        int i;
        if ( key [ i = locate(s) ] == 0 )    // new string
        {
            key[i] = buffer + buf_end;
            strcpy ( buffer + buf_end, s );
            buf_end += strlen ( s ) + 1;
        }

        return ( val[i] );
    }
    ```

- There is no operator `[][]` for double subscripted arrays

Type conversions

- Essential for effective operator overloading
- Conversion from type `X` to type `Y` can be achieved simply by supplying a constructor for `Y` with argument `X` or (`X&`)

  ```
  complex ( double )
  string ( char *)
  fraction ( int )
  ```

- Cannot be used to convert back to built in types (they are not classes)
  - We can circumvent this restriction by using a member function, for example

    ```
    class fraction
    {
        int num, denom;
        public:
            // ...
            operator double();
    };

    fraction::operator double ( void )
    {
        return ( double ( num ) / double ( denom ) );
    }
    ```

- Type conversion/promotion does not work across more than one level of user-defined type conversion when trying to match an overloaded function
  - The following will not work

    ```
    fraction f ( 1, 2 );
    complex z ( 2, -1 );
    complex w = f * z;
    ```

  - We can help the compiler by

    ```
    complex w = double ( f ) * z;
    ```

- Type conversion and reference arguments
  - Let us look at the swap function again

    ```
    void swap ( double& a, double& b )
    {
        double tmp = a;
        a = b;
        b = tmp;
    }
    ```

  - Now consider the following code

    ```
    double x = 3.0;
    fraction f ( 1, 2 );
    swap ( x , f );  // f is not changed
    ```

- The following sequence of steps takes place
  - ∗ A type conversion `fraction` → `double` is performed
  - ∗ Result is stored in a temporary variable
  - ∗ A reference to the temporary variable is passed to `swap` (instead of `f`)
  - ∗ The contents of the temporary variable are swapped with `x`
  - ∗ The temporary variable is destroyed
  - ∗ `f` remains unaffected

- Unintended type conversion
  - C++ automatically uses constructors with a single argument as type converters
  - Look for the error in the following code

```
class point
{
    double _x, _y;
    public:
        point ( double x = 0, double y = 0 );  // Constructor
        // ...
};

main()
{
    double a, r, x, y;
    // ...
    point p = ( x + r * cos ( a ), y + r * sin ( a ) );
    // ...
}
```

  - The intention was to write

```
point p ( x + r * cos ( a ), y + r * sin ( a ) );
```

  - The code compiles and runs using the comma in the expression as the comma operator
  - The end result is

```
point p ( y + r * sin ( a ), 0 );
```

  - This happened because default arguments can lead to unintended results

- Finally, never ever forget the precedence of overloaded operators