

10.1 More on Classes

The C++ language also has some new features that enhance the readability of your code. In the next program pass by reference (without using pointers) and function and operator overloading are shown as well as friend functions.

Consider the following program:

```

#include "iostream.h"
#include "string.h"

class Person
{
private:
    int    age;
    char   *name;
public:
    Person(int a, char *s);
    ~Person( );
    void   display( );
    void   assign(int a, char *s);
    void   operator = (Person &other_object);
    friend void show_older(Person &a, Person &b);
};

Person::Person( int a, char *s)
{
    cout << "Creating " << s << "\n";
    age = a;
    name = new char[80];
    strcpy (name,s);
}

Person::~~Person()
{
    cout << "Deleting " << name << "\n";
    delete name;
    name = NULL;
}

void
Person::assign( int a, char *s)
{
    age = a;
    if (name == NULL)
        strcpy (name,s);
}

void
Person::display()
{
    cout << name << " is " << age << " years old \n";
}

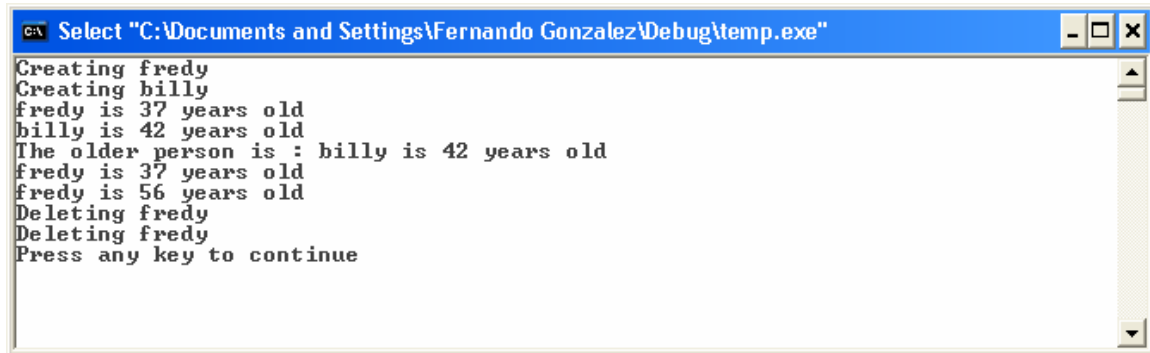
```

```
void
Person:: operator = (Person &other_object)
{
    age = other_object.age;
    strcpy(name,other_object.name);
}

void
show_older(Person &a, Person &b) // I am a friend of Person
{
    // so I can access private data
    cout << "The older person is : ";
    if (a.age > b.age)
        a.display();
    else
        b.display();
}

void
main()
{
    Person fred(37 ,"fredy"), bill(42, "billy");
    fred.display();
    bill.display();
    show_older(bill,fred);
    bill = fred;           // Our "=", not the usual one.
    bill.display();
    bill.assign(56, "new guy");
    bill.display();
}
```

The output is:

A screenshot of a Windows command prompt window. The title bar reads "c:\ Select 'C:\Documents and Settings\Fernando Gonzalez\Debug\temp.exe'". The window contains the following text:

```
Creating fredy
Creating billy
fredy is 37 years old
billy is 42 years old
The older person is : billy is 42 years old
fredy is 37 years old
fredy is 56 years old
Deleting fredy
Deleting fredy
Press any key to continue
```

The line

```
name = new char[80];
```

in the constructor ask the operating system for memory to implements its string. The constructor accepts an integer and a pointer to a string (an array). The constructor creates a new array by asking the operating system to give it sufficient memory to implement the array (dynamic memory allocation). Then each component in the input string is copied to the newly allocated string.

The line

```
delete name;
```

in the destructor returns the memory back to the operating system.

The member function called assign allows the user to assign values to the object after instantiation. Notice it does not get new memory since the object already has memory.

The line

```
void display( );
```

declares a member function called display(). It is the only way to view the private elements of the object.

The member function

```
void
Person::operator = (Person &other_object)
{
    age = other_object.age;
    strcpy(name, other_object.name);
}
```

```
}
```

demonstrates the use of operator overloading and the new pass by reference in C++.

First the pass by reference. By putting a & sign in front of an argument in the declaration of a function it tells the compiler that that variable is to be passed by reference. The compiler then puts in the necessary pointers to get it to work. The programmer write the code as though it is passed by value (no pointers stuff) and the compiler fixes it automatically.

For example both programs do the same thing:

<pre>void bill(int *x) { *x = *x + 5; } main() { int y = 1; bill (&y); printf("y is %d\n",y); }</pre>	<pre>void bill(int &x) { x = x + 5; } main() { int y = 1; bill (y); printf("y is %d\n",y); }</pre>
--	---

The output for both is:

```
y is 6
```

Note in the right version the, we write the code as though it were pass by value. In reality the compiler puts in all of the pointer stuff and produces machine code equivalent to the program on the left (with pointers).

The next new feature is the operator overload. The line

```
void
Person::operator = (Person &other_object)
```

Indicated the function is to be called “=”. There are several things happening here.

First the function is called “=” not “bill”, “copy”, etc but a symbol. Next the name of “=” is already used. Finally the way we call the function also changes.

When we call the function we use:

```
bill = fred;           // Our "=", not the usual one.
```

Not

```
= (bill, fred);
```

as we for other functions as in

```
copy (bill, fred);
```

This makes the code much nicer to read. It looks very mathematical.

Next since we have functions = for integers, double, char, etc. we are basically using an already existing name. In the line

```
int x, y;  
x = y
```

the = is a function that copies the data in the right parameter (the y) to the left parameter (the x). If x and y were type double then the = function will be a different function that accepts doubles. We are using operator overloading. Operator overloading like function overloading is the act of giving more than one function the same name. The compiler distinguishes between them by the argument list. For example

```
int bill(int x)
{
    return x + 5;
}

double bill(double x)
{
    return x + 5.5;
}

main()
{
    int    a = 1;
    double b = 1.5;

    bill(a);
    bill(b);
    printf("a is %d and b is %lf\n", a, b);
}
```

The output is

```
a is 6 and b is 7.0
```

In this program

```
bill (a)
```

calls the top bill function and

```
bill(b);
```

calls the bottom one. The compiler knows which one to call by the argument passed to it.

So the line

```
bill = fred;           // Our "=", not the usual one.
```

In our program calls the = function that we wrote for the class. The compiler knows to call this function because of the type of bill and fred being object of type Person.

The member function called operator = is used to allow the assignment of objects. If one does not supply this function, assignment of objects may not be possible or the compiler may generate a default function. If a default function is used it will only copy the contents of the source object to the destination object. If the source object contains a pointer then the default assignment function will only copy the pointer itself and not the contents of the memory that the pointer points to. This could lead to problems. It is advised that if you plan to use an operator with an object that you specify the member function and do not rely on the default.

The line

```
friend void show_older(Person &a, Person &b);
```

in the class declaration says that while show_older() is not a member function of the class it does have access to its private elements. Now the function:

```
void
show_older(Person &a, Person &b) // I am a friend of Person
{
    // so I can access private data
    cout << "The older person is : ";
    if (a.age > b.age)
        a.display();
    else
        b.display();
}
```

receives 2 objects and can access its private elements. From the code you can see that object a and b are treated the same. This shows that this is a function involves a and b but is not part of either. In the function call:

```
show_older (bill, fred) ;
```

bill and fred are treated the same as well. If it not were for friend functions this is how it will have been written:

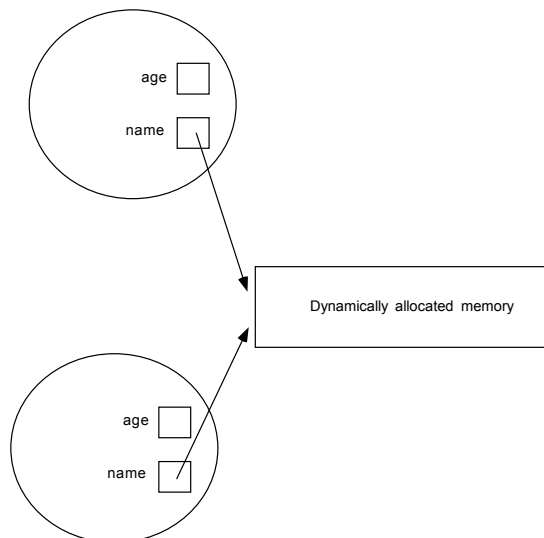
```
bill.show_older (fred) ;
```

or

```
fred.show_older (bill) ;
```

This indicates the function corresponds to bill or fred when it corresponds to neither.

In both functions that have an object being passed, the object is passed by reference. Passing by value could lead to the following situation. An object is declared in the main function. The constructor allocates memory to this object. The object is passed to the function as pass by value. The function creates a new object, the objects constructor is executed and then copies the contents of the object being passed to the new object. The function you created to copy objects is not called instead each variable is copied. When the pointer variable called name is copied it only copies the pointer itself not the contents of the memory that name points to. This means the name variable in the new object points to the same memory as the old object.



This is pretty bad but it gets worse. When the function terminates it deletes all of its local variables. Included is the object it created. When this object is destroyed its destructor is called. The destructor then returns the memory pointed to by name back to the operating

system. After the function is called the object that was passed has lost its memory even though name still points to it. To fix this it is best to always pass objects by reference.