Overloading the Assignment Operator

- Remember, C++ will define a default and naïve copy constructor for you if you don’t provide one.
- It will just copy member variables (potential for dangling pointers)
- In the case of Course, we’d need to override the default copy constructor to make sure the storage was copied properly.

```cpp
Course::Course(Course &aCopy)
{
    // Copy storage into new instance if necessary...
}
```

Again, this will take care of the case where someone tries to assign to a Course variable when it is declared:

```
Course newCourse = anotherCourse;
```

Overloading the Assignment Operator

- Oh, yeah… we should always make sure that our source and destinations aren’t the same...
- We do this by adding the following code:

```cpp
Course &Course::operator=(Course &argCourse)
{
    // Make sure we actually have two different pointers
    if (this != &argCourse)
        duplicate(argCourse);
    return *this; // Huh?
}
```

```
What is “this”, anyway?
This is a pointer to the current instance of the class we are in.
```

Demonstration #1

Overloading the Assignment Operator

```cpp
Course &Course::operator=(Course &argCourse)
{
    // Assume we have a member function called duplicate()
    // which copies values from the Course passed in into
    // our instance.
    duplicate(argCourse);
    return *this; // Huh?
}
```

- Remember that when overloading the = operator you are going to be assigning to an existing instance. If that instance has dynamically allocated data it should be freed.
- We return a reference so that c1 = c2 = c3 works...

```
```
More About Types
- Do you know how to write a type name?
- There is a simple convention for writing a type name...
- Start with a variable declaration of the desired type, then remove the variable...

```c
int k; // Type is really just "int"
int *k; // Type is really just (int *)
int *k[]; // Type is really just (int *) *
```

Remember, the asterisk binds tighter than the square brackets
Another way to "define" our own user types is through the typedef keyword.
This is a way of creating more of a "shorthand" for existing types rather than actually defining a new type.

typedef
- When using typedef to define a shorthand for some array type, place the right brackets just to the right of the name chosen for the new type.
- Consider a new type called String255 which is an array of 255 characters (well, plus 1 to account for the NULL byte)

```c
// Define a type to represent C style strings of 255
// characters (or less). Leave an extra byte for the NULL
// terminating byte.
typedef char String255[256];
```

OK, let's take a look at some of this in action...

```c
typedef
- Consider a new type named StringArray which defines an array of Str255 types:
- It could either be defined as a pointer or an array itself

```c
// Define a type to represent C style strings of 255
// characters (or less). Leave an extra byte for the NULL
// terminating byte.
typedef String255 *StringArray; // arbitrary size
typedef String255 StringArray15[15]; // 15 String255's
```

Type Equivalence
- If two types are equivalent they can be assigned to each other without needing to have a specially overloaded assignment operator.
- Two types are equivalent if they have the same name
  - Remember, typedefs don’t define new types, just provide shortcuts

```c
typedef Student *StudentPtr;
typedef Student *UndergradPtr;
// Both StudentPtr and UndergradPtr are equivalent.
StudentPtr oneStudent;
UndergradPtr anotherStudent = new UndergradPtr();
oneStudent = anotherStudent; // this is legal because they
// are type equivalent
```
Sizeof operator
- The size (in bytes) that any data type takes up may be retrieved by the user by calling the sizeof function.
- In C++, this information is really only useful if you are writing an alternative to new.

For some structures/classes sizeof() might return a value larger than the sum of all fields in question (padding).

Type Conversions
- Early on we touched on the issue of type conversions.
- When assigning between two different types (especially numeric) C++ will do its best to implicitly convert between the type you are assigning from to the type you are assigning to.

```
int main()
{
    cout << "sizeof(int) is " << sizeof(int) << endl;
    cout << "sizeof(float) is " << sizeof(float) << endl;
    cout << "sizeof(Course) is " << sizeof(Course) << endl;
    return 0;
}
```

Demonstration #3
Implicit Type Conversions (Numeric)

Type Conversions
- What about non-numeric types?
- Well you can convert between pointers and integers and between pointers to different types...
- But you need to typecast them, like this:

```
int main()
{
    Control *ctrl1 = new PopupMenu(5,5,100,20,"My Menu");
    PopupMenu *pm;
    pm = ctrl1; // No, the compiler won't let you do this!
    pm = (PopupMenu*)ctrl1; // But this is ok...
}
```

Type Conversions
- Typecasting can be a powerful tool, especially when dealing with derived classes needing to be accessed from a base class pointer.
- Consider the following pseudo-code...

```
// The following is pseudo-code, it is not complete...
int main()
{
    MenuObject *itemList[50];
    itemList[0] = new MenuItem(); // Assume constructors
    itemList[1] = new SubMenu();
    // Now, typecast our way to the derived classes
    ((MenuItem*)itemList[0])->setCmd(....);
    ((SubMenu*)itemList[1])->appendItem(....);
}
```
### Type Conversions
- The moral of the story is to be very, very careful with typecasting.
- Essentially, it overrides the compiler’s type checking mechanism.
- So you can do some pretty bizarre things.
- But, used responsibly, you can do useful things as well.
- Did you know that you can define what it means to typecast an instance/reference to a class you’ve defined?
- Consider the following code...

```cpp
#include <iostream>

class X { private:
  int a,b;
};

public:
{
  class Y
};

INT myInt(4);
{
}

int main()
{
  INT myInt(4);
  int x = myInt // Compiler won’t like this!
}
```

### Private Inheritance
- When looking at inheritance, we’ve always used a declaration of the following form:

```cpp
class X : public Y {
  // New code here...
};
```

- I’ve asked you to take it on faith that `public Y` is simply the syntax you must use to say that the class `X` is derived from class `Y`.
- Now, consider the following partial definitions of `X` and `Y`...

```cpp
class Y {
  public:
    int a,b;
  protected:
    int c,d;
};

class X : public Y {
  public:
    int h,i;
};
```

- As we’ve gone over before, a member function in class `X` has access to member variables `a,b,c,d,h,i` from the base class `Y`.
- When working with `X` outside of the class, `a,b,h,i` are accessible.

### Private Inheritance
- That is to say that no members from a privately inherited base class may be accessed from outside the scope of the derived class.
- It also means that the relationship `X` is a `Y` or `X` is a kind of `Y` doesn’t really hold up here.
- Why? Because if `a Y` has certain public methods and `X` is a `Y`, then `X` should have the same public methods.
- With private inheritance this is not the case, as the public methods in `Y` are not public methods in `X`.
- So, then, what is private inheritance good for?
- LMG suggests the following (from pg. 231):
  - This is what private inheritance is for. If A is to be implemented as a B, and if class B has virtual functions that A can usefully override, make A a private base class of B.
  - Is this really useful? Wait… before you decide...

### Type Conversions
- We could just use the `INT::getValue()` to make the compiler happy, but there’s a better way.
- We can overload the `(int)` typecast in `INT`...

```cpp
INT::operator int() const
{
  return value;
}
```

- Now, the following code will compile:

```cpp
int main()
{
  INT myInt(4);
  int x = myInt // Now, compiler is happy!
}
```
Private Inheritance

A pointer to a class derived privately from a base class may not be assigned directly to a pointer to the base class.

```cpp
class X {
public:
    void doSomethingUseful(int); // arbitrary public member func
    int a,b;
};

class X:private Y {
public:
    int h,i;
};
```

- Ouch! We've been able to do this before but now the compiler won't let us :-(.
- Oh, wait, this is C++, it will let me do almost anything with a little "coercion"

```cpp
X x; // declare an instance of the class X
Y *yPtr; // pointer to an instance of base class
yPtr = &x; // COMPILER ERROR. access violation
```

- Private Inheritance

```cpp
int main() {
    X anX;
anX.doSomethingUseful(); // access violation
    return 0;
}
```

- Attempting to access doSomethingUseful() from a variable of class X is an access violation.
- That's because doSomethingUseful() is a public method of a privately inherited base class.
- Bummer.
- Oh, wait, this is C++... Can I coerce my way around this restriction?
- You betcha!

```cpp
X x; // declare an instance of the class X
Y *yPtr; // pointer to an instance of base class
yPtr = (Y *)&x; // Oh, ok, this is fine
```

- Since doSomethingUseful() is a public member of Y, it is not accessible outside of the scope of X. That is...

```
Demonstration #4
```

- The addition of the line Y::doSomethingUseful; into the public section of the class definition of x solved our problem.
- Notice that only the name of the member function from the base class to be made public is used, there is no parameter list.
- So, back to my original question... Is it useful?
- Let's see it in action first...
Private Inheritance--Is it useful?
- Well, put a feature in a language and someone will find a way to use it!
- In my programming travels I have never seen it used.
- It's not for doing straight inheritance, because the relationships break down.
- It's not for doing interfaces since the pure virtual approach is much cleaner, simpler, and enforces the implementation of all members.
- The example given in the books shows a case where a generic base class is used to provide functionality to a more specific derived class.
  - the derived class is conceptually different from the base class
  - List vs. Stack
  - Many of the features of the base class might not be applicable to the derived class.
  - Removing the nth element of a list is not a stack-like function.
  - In this case, private inheritance may be appropriate.