# The Constant Keyword

The const Keyword, Reading

This may not have been covered in your prerequisite courses, but it's something we'll need pretty soon in this course, so we might as well take care of it right now. It's the "const" keyword. If you already completed, Comsc-200, then you know exactly what we're going to say in this reading.

In any case, you already know about this in connection with declaring constants, like this:

  const int CAPACITY = 100;  
  const double PI = 3.14159265;

But const is also used in other contexts in C++, some of which we learn about now, leaving one more for later.

Read-Only Pointers

By default, pointers can be used to access the value stored at the memory location they store. With that access, the can both "look" at the value, and they can reassign another value to that location, like **\*p = 100;**, using "dereferencing" with the "leading star" symbol.

But there's another type of pointer that cannot be used to reassign. It's got less privilege than an ordinary pointer. Why would we ever care to use something in code that does less than something else? There's two answers -- first, programmers like to apply what's called the "principle of least privilege" to lessen their chances of logic errors by converting them from logic error into syntax errors that the compiler can catch. The other is that some elements of the C and C++ libraries expect you to use read-only pointers with them. Here's how they are declared:

  const int\* p;

You might expect this to work like the const we already know about -- to make it so the value stored in p can never change. But not so! p can still be reassigned. What it does mean is that p cannot be used in statements like **\*p = 100;**. It's for looking only -- not for touching! We'll use it for traversing linked lists:

  Student\* firstStudent = 0; // head pointer  
  
  for (**const** Student\* p = firstStudent; p; p = p->next)  
  {  
    cout.width(30);  
    cout << p->name;  
    cout.width(6);  
    cout << p->age;  
    cout.precision(2);  
    cout.width(8);  
    cout << p->gpa << endl;  
  }

An **alternate syntax** for a read-only pointer repositions the const keyword after the data type, but still before the star:

  for (Student const \*p = firstStudent; p; p = p->next)

Read-Only Reference Variables

Reference variables show up mostly as function parameters. You may have learned to share an object with a function like this:

void fun(Student);  
  
int main( )  
{  
  Student x;  
  fun(x);  
}  
  
void fun(Student s) // s is a copy of x in main  
{  
  ...  
}

That makes a copy of the object in main for use in the function. Using the trailing ampersand symbol, you can share the original and avoid having to make a copy:

void fun(Student&);  
  
int main( )  
{  
  Student x;  
  fun(x);  
}  
  
void fun(Student& s) // s is an alias for x in main  
{  
  ...  
}

The only issue here is that the function now "knows where x lives" and knowing that, it can make changes. That's okay, if the "fun" function is supposed to do that. But if the function only needs to "look" at the object, the programmer should "protect" the object from modification inside the function. That's where the const keyword comes in:

void fun(**const** Student&);  
  
int main( )  
{  
  Student x;  
  fun(x);  
}  
  
void fun(**const** Student& s) // s is an alias for x in main  
{  
  ...  
}

This is what we'll use most often in this course for objects as function parameters, because (1) it's more efficient to send the memory location of an object than to make a copy of the object, and (2) we still want to protect the original object from modification inside the function.