

ENGG 233 Fall 2004 L02 (Lecture Section 02): Code examples for Sept. 28

Author: Dr. S. A. Norman

Electronic copies of handouts for L02 can be found at

<http://www.enel.ucalgary.ca/People/Norman/engg233-L02-fall2004/>

```
// pass-by-value.cc (example for Sept 28 2004)
// Demonstrates a logic error due to not understanding pass-by-value.
#include <iostream>
using namespace std;

const double MATH_CONSTANT_PI = 3.1415926535897931;
void rad2deg(double x);

int main()
{
    double v;
    cout << "Please enter a number of radians: " << endl;
    cin >> v;
    cout << "You entered " << v << " radians." << endl;
    rad2deg(v);
    cout << "That is " << v << " degrees." << endl;
    return 0;
}

void rad2deg(double x)
{
    x = x * 180.0 / MATH_CONSTANT_PI;

    // point one

    return;
}
```

```
// pass-by-ref.cc (example for Sept 28 2004)
// One way to fix the logic error in pass-by-value.cc.
// (A different fix would be to have rad2deg return a value).
#include <iostream>
using namespace std;

const double MATH_CONSTANT_PI = 3.1415926535897931;
void rad2deg(double& y); // ATTENTION: Note presence of &

int main()
{
    double v;
    cout << "Please enter a number of radians: " << endl;
    cin >> v;
    cout << "You entered " << v << " radians." << endl;
    rad2deg(v);
    cout << "That is " << v << " degrees." << endl;
    return 0;
}

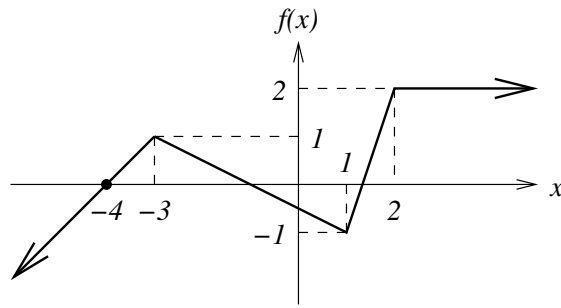
void rad2deg(double& y) // ATTENTION: Note presence of & again
{
    y = y * 180.0 / MATH_CONSTANT_PI;

    // point two

    return;
}
```

Example of multiway decision-making: Consider the mathematical function given below, with its graph shown to the right.

$$f(x) = \begin{cases} x + 4 & \text{if } x < -3 \\ -0.5x - 0.5 & \text{if } -3 \leq x < 1 \\ 3x - 4 & \text{if } 1 \leq x < 2 \\ 2 & \text{if } x \geq 2 \end{cases}$$



A simple way to code this function using plain `if` statements is shown to the right. It produces the right result, but it's a bit inefficient. For example, if x is less than -3.0 there is no reason to check all the conditions in the last three `if` statements.

```
double f(double x)
{
    double result;
    if (x < -3.0)
        result = x + 4.0;
    if (-3.0 <= x && x < 1.0)
        result = -0.5 - 0.5 * x;
    if (1.0 <= x && x < 2.0)
        result = 3.0 * x - 4.0;
    if (x >= 2.0)
        result = 2.0;
    return result;
}
```

The code to the right is more efficient. The idea is first to write an `if-else` statement to choose between $x < -3$ and `not(x < -3)`. Inside the `else` part of that, we write another `if-else` statement to choose between $x < 1$ and `not(x < 1)`. Finally the innermost `if-else` statement chooses between $x < 2$ and `not(x < 2)`.

```
double f(double x)
{
    double result;
    if (x < -3.0)
        result = x + 4.0;
    else
        if (x < 1.0)
            result = -0.5 - 0.5 * x;
        else
            if (x < 2.0)
                result = 3.0 * x - 4.0;
            else
                result = 2.0;
    return result;
}
```

This kind of structure is sometimes called a *nested if-else* statement, because it has `if-else` statements nested inside other `if-else` statements.

Now consider the code on the right. To a C++ compiler, it is identical to the preceding version; it differs only in its layout, and for the most part C++ compilers do not care about layout.

```
double f(double x)
{
    double result;
    if (x < -3.0)
        result = x + 4.0;
    else if (x < 1.0)
        result = -0.5 - 0.5 * x;
    else if (x < 2.0)
        result = 3.0 * x - 4.0;
    else
        result = 2.0;
    return result;
}
```

However, this is the preferred style for a nested `if-else` statement, because most C++ programmers would find it nicer looking and easier to read. It's clearer that this code is choosing exactly one of four alternatives.