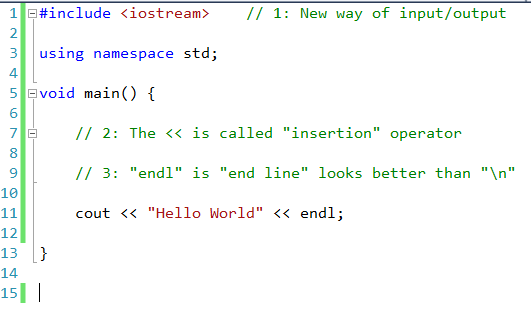
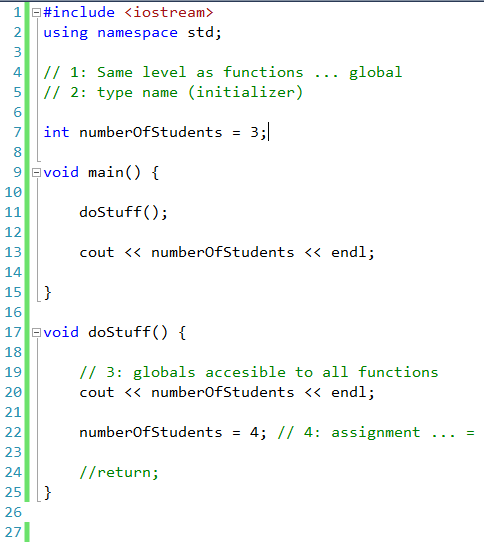
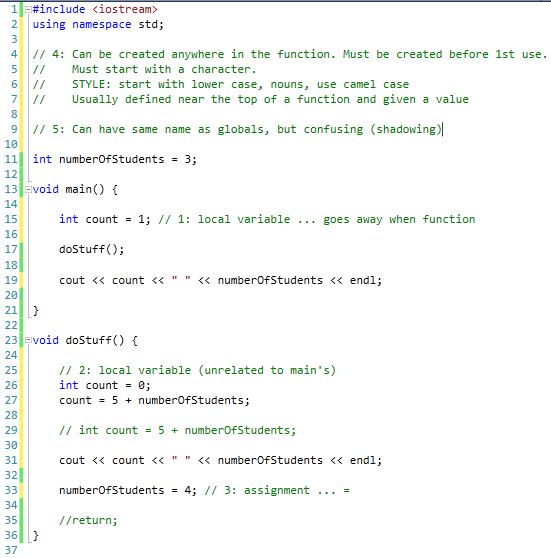
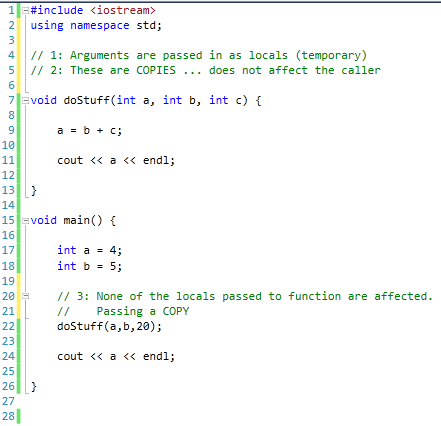
# Variables and Expressions

New way of input/output in C++. Uses operator overloading (in a few days).









# Types and Sizes

#include <iostream>

using namespace std;

#include <climits>

void main() {

// 1: Examples of types (page 72)

char a; // at least 8 bits (not unicode)

short b; // at least 16 bits

int c; // at least as big as a short

long d; // at least 32 bits and at least as big as an int

long long e; // at least 64 bits and at least as big as a long

float f; // unspecified (usually 32 bits)

double g; // unspecified (usually 64 bits)

long double h; // unspecified (usually 128 bits)

cout << sizeof(double) << endl; // 2: "sizeof" evaluated at compile time

cout << INT\_MAX << endl; // 3: Using constants from climits

}

The size of “int” is different for different machines. The size is what is “natural” for the CPU architecture. Your code might work on one machine but not another.

The upper bit of a value is its sign bit. You can use “unsigned” in front of a type to treat the value as unsigned.

When math overflows the storage size of a value it will wrap. Very negative numbers will suddenly be very positive.

#include <iostream>

using namespace std;

void main() {

unsigned int x = 0;

x = x - 1;

cout << x << endl;

// 0000

// - 0001

// ----

// 1111 (borrow is 1)

}

# Literals

void main() {

float f = 1/3;

//float f = 1.0/3.0 1: 1 is integer, 1.0 is float

cout << f << endl; // 1: What gets printed here?

}

You can add a postfix to constants to tell the compiler how to treat them.

void main() {

int a = 123; // Nothing is "int"

long b = 123L; // L is "long", UL is "unsigned long"

float c = 3.14F; // F is "float"

double d = 2.0L; // L on a "float" is a double

char e = 'Z'; // In ticks ... character literal

char \*p = "HELLO"; // String constants

p = "HELLO \"WORLD"; // Quotes within quotes are "escaped" ... page 85

cout << p << endl;

}

The compiler will convert between types for you automatically, but data could be lost. The compiler will give you a warning unless you explicitly cast.

void doStuff(int x) {

cout << x << endl;

}

void main() {

float a = 1.0;

doStuff(a); // warning C4244: 'argument' : conversion from 'float' to 'int', possible loss of data

doStuff((int)a);

}

# Boolean Values

C++ has a Boolean type that uses keywords “true” and “false”. Expressions still use the old “0 means false and anything else is true”.

#include <iostream>

using namespace std;

#include <climits>

void main() {

bool a = true;

bool b = false;

cout << a << " " << b << endl;

}

They are still treated as numerical (0 and 1) at runtime.

# Operators

void main() {

int a = 2+3\*4-2;

// int a = 2+(3\*4)-2; // 1: Use parenthesis. They are free.

cout << a << endl;

}

Look at these in code:

+, -, \*, / Familiar math operations

% Modulo … remainder of integer division

&, |, ~, ^ Bitwise AND, OR, NOT, XOR

<<, >> Bitwise shifts

++, -- Increment and decrement

void main() {

int a = 4;

//int b = ++a;

int b = a++;

// a++ In a standalone the ++a and a++ are the same

cout << a << " " << b << endl;

}