Variables, operators and mathematical functions

Anna Simon

August 28, 2015

Office hours:
Monday 3pm-4pm and Tuesday 4pm-5pm.
Room 221 NSH

Homework:

- slides from each week and a complete list of assignments will be posted on sakai on Fridays by the end of the day,
- homework due at noon Wednesday,
- submit homework via sakai, the file name should contain your last name and the number of the assignment (e.g. simon_3.4.C),
- read the instructions in syllabus before submitting your code.

Variables: declarations

How to declare

```
bool bValue;
char chValue;
int nValue;
float fValue;
double dValue;
```

Assignment

```
int nValue = 5; // explicit assignment
int nValue(5); // implicit assignment
```

Even though implicit assignments look a lot like function calls, the compiler can resolve them properly. However, it might look confusing for a user.

Declaring multiple variables

```
int nValue1, nValue2;
```

```
int nValue1;
int nValue2;
```

Variables of the same type can be declared and initialized within the same line, separated by comma:

```
int nValue1 = 5, nValue2 = 6;
int nValue3(7), nValue4(8);
```

```
int nValue1 = 5;
int nValue2 = 6;
int nValue3 = 7;
int nValue4 = 8;
```

Keywords and naming identifiers

Keywords are reserved and cannot be used as identifiers:

alignas (since C++11)	else	requires (concepts TS)
alignof (since C++11)	enum	return
and	explicit	short
and_eq	export(1)	signed
asm	extern	sizeof
auto(1)	false	static
bitand	float	static_assert (since C++11)
bitor	for	static_cast
bool	friend	struct
break	goto	switch
case	if	template
catch	inline	this
char	int	thread_local (since C++11)
char16_t (since C++11)	long	throw
char32_t (since C++11)	mutable	true
class	namespace	try
compl	new	typedef
concept (concepts TS)	noexcept (since C++11)	typeid
const	not	typename
constexpr (since C++11)	not_eq	union
const_cast	nullptr (since C++11)	unsigned
continue	operator	using(1)
decltype (since C++11)	or	virtual
default(1)	or_eq	void
delete(1)	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
	reinterpret_cast	xor_eq

Conventions when naming identifiers:

- The identifier can not be a keyword. Keywords are reserved.
- The identifier can only be composed of letters, numbers, and the underscore character. That means the name can not contains symbols (except the underscore) nor whitespace.
- The identifier must begin with a letter or an underscore. It can not start with a number. Variable names should start with a letter.
- C++ distinguishes between lower and upper case letters. `nvalue` is different than `nValue` is different than `NVALUE`.

Hungarian notation

Hungarian notation is a naming convention in which the type and/or scope of the variable is used as a naming prefix for that variable.

```
int value; // non-Hungarian
int nValue; // the n prefix denotes an integer

double width; // non-Hungarian
double dWidth; // the d prefix denotes a double
```

Most common prefixes:

Type prefix	Meaning	Example
b	boolean bool	bHasEffect;
c (or none∗)	class	Creature cMonster;
ch	char (used as a char)	char chLetterGrade;
d	double, long double	double dPi;
f	float	float fPercent;
n	short, int, long char used as an integer	int nValue;
s	struct	Rectangle sRect;
str	C++ string	std::string strName;

Integers: overflow

Overflow

Let's assume 4-bit representation of an integer.

decimal value	binary representation
0	0000
1	0001
7	0111
15	1111

What happens when a value greater than 15 is assigned to this variable? E.g. 16, in binary form: 10000. But only the last four bits are saved to the variable, so now we have zero instead of 16. This is called **overflow**.

Integer division

What is the result of the following expression?

```
int x = 4 / 3;
```

How to fix it, to obtain the correct result?

Floating point number: precision

Precision of cout

What is the output of this programs?

```cpp
#include <iostream>
using namespace std;

int main()
{
    float fValue;
    fValue = 1.222222222222222f;
    cout << fValue << endl;
    fValue = 111.22222222222222f;
    cout << fValue << endl;
    fValue = 111111.222222222222f;
    cout << fValue << endl;
    return 0;
}
```

```cpp
#include <iostream>
#include <iomanip> // for setprecision()
int main()
{
    cout << setprecision(16); // show 16 digits
    float fValue = 3.33333333333333333333333333333333333333f;
    cout << fValue << endl;

    double dValue = 3.3333333333333333333333333333333333333;
    cout << dValue << endl;

    fValue = 123456789.0f;
    cout << fValue << endl;
return 0;
}
```

Floating point number: rounding error

What is the output of the following programs?

```
#include <iomanip>
using namespace std;

int main()
{
    cout << setprecision(17);
    double dValue = 0.1;
    cout << dValue << endl;
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setprecision(17);
    double dValue;
    dValue = 0.1 + 0.1 + 0.1 +
         0.1 + 0.1 + 0.1 + 0.1 +
         0.1 + 0.1 + 0.1; //
         nine additions
    cout << dValue << endl;
}
```

Comparison of floating point numbers

Compare the outputs of the following codes

```cpp
int x = 5; // integers have no precision issues
if (x==5)
    cout << "x is 5" << endl;
else
    cout << "x is not 5" << endl;
```

```cpp
float fVal1 = 1.345f;
float fVal2 = 1.123f;
float fTotal = fVal1 + fVal2; // should be 2.468

if (fTotal == 2.468)
    cout << "fTotal is 2.468";
else
    cout << "fTotal is not 2.468";
```

Summary

- Floating point numbers offer limited precision.
  floats: 7 significant digits
  doubles: 16 significant digits
  (Note: placeholder zeros do not count as significant digits, so a number like 22,000,000,000, or 0.00000033 only counts for 2 digits).
- Floating point numbers often have small rounding errors.
  Comparisons on floating point numbers may not give the expected results when two numbers are close.

Type conversion

Let's examine the following lines of a code:

```
double dValue = 3; // implicit conversion to double value 3.0
int nValue = 3.14156; // implicit conversion to integer value 3
```

How will the following expression be evaluated: `2 + 3.14159`?

Hierarchy of data types

The compiler will "upgrade" the type of literal constant that is lower in the hierarchy to match the other type.

- Long double (highest)
- Double
- Float
- Unsigned long int
- Long int
- Unsigned int
- Int (lowest)

Warning

How will this expression be evaluated: `5u - 10`?

Type casting

When using variables the types will not be automatically converted.

```
int nValue1 = 10;
int nValue2 = 4;
float fValue = nValue1 / nValue2;
```

However programmer can manually "upgrade" the type of a variable:

```
int nValue1 = 10;
int nValue2 = 4;
float fValue = (float)nValue1 / nValue2; //one variable CASTED as float, the second gets upgraded and the result is float
```

But this may sometimes be misused, e.g. get rid of `const`.

`static_cast` is safer, allows only standard type conversions.

```
int nValue1 = 10;
int nValue2 = 4;
float fValue = static_cast<float>(nValue1) / nValue2;
```

This can also be used to tell the compiler that we intend to do something considered dangerous. Compiler will complain about this:

```
int nValue = 100;
nValue = nValue / 2.5; //conversion from float to int
```

But one can tell the compiler that he/she means it:

```
int nValue = 100;
nValue = static_cast<int>(nValue / 2.5);
```

Variables: local variables

A **compound statement** (block) is a group of statements to be executed placed within curly brackets ({}).
A variable that is declared within a block (**local variable**) is accessible only within that block and is destroyed when the end of the block is reached.

```
int main()
{
    using namespace std;
    cout << "Enter a number: ";
    int nValue;
    cin >> nValue;

    if (nValue > 0)
    { // start of nested block
        cout << nValue << " is a
            positive number" <<
            endl;
        cout << "Double this number
             is " << nValue * 2 <<
             endl;
    } // end of nested block
return 0;
}
```

```
int main()
{
    int nValue = 5;

    { // begin nested block
        double dValue = 4.0;
    } // dValue destroyed here

    // dValue can not be used here
        because it was already
        destroyed!

    return 0;
} // nValue destroyed here
```

Variables: global variables

Global variables have program scope, they can be accessed from anywhere within the code.

```cpp
int g_nX; // global variable

int main()
{
    int nY; // local variable nY

    // global vars can be seen everywhere in program
    // so we can change their values here
    g_nX = 5;
} // nY is destroyed here
```

They can be used across programs with multiple files (declared in one of the files and then redeclared as `extern` in the other files.)

`global.cpp`:

```cpp
// declaration of g_nValue
int g_nValue = 5;
```

`main.cpp`

```cpp
// extern tells the compiler this variable is declared elsewhere
extern int g_nValue;

int main()
{
    g_nValue = 7;
    return 0;
}
```

Variables: global scope

Global variables are dangerous! They make the program complex, their value can be changed by any function that is called, which makes them difficult to follow.

```cpp
// declare global variable
int g_nMode = 1;

void doSomething()
{
    g_nMode = 2;
}

int main()
{
    g_nMode = 1;
    doSomething();

    // Programmer expects g_nMode to be 1
    // But doSomething changed it to 2!

    if (g_nMode == 1)
        cout << "No threat detected." << endl;
    else
        cout << "Launching nuclear missiles..." << endl;
    return 0;
}
```

struct: declaration

C++ allows for creating aggregated data type that groups multiple variables together using `struct`. Example: define a data structure for a segmented detector:

```
struct detector_t
{
    float energy;
    float time;
    int ringID;
    int sectorID;
};

detector_t DE;
detector_t E1;
```

This way multiple variables of the same struct type can be defined (e.g. DE, E1). The struct members can be accessed as regular variables:

```
float totalEnergy;
if (DE.ringID == E1.ringID)
{
    totalEnergy = DE.energy + E1.energy;
}
```

struct

The whole struct can be passed to a function (the struct has to be delclared as a global type):

```
void PrintData(detector_t detector)
{
    cout << "Energy: " << detector.energy <<endl;
    cout << "Time: "  << detector.time << endl;
}

int main()
{
    detector_t DE;
    DE.energy= 1000.;
    DE.time= 104238742356.;
    DE.ringID=3;
    DE.sectorID=6;

PrintData(DE);

return 0;
}
```

structs can be nested:

```
struct telescope_t
{
    detector_t DE;
    detector_t E1;
    int numberOfDetectors;
};

int main()
{
    telescope_t telescope;

    telescope.DE.energy = 1000.;

}
```

struct: initalization

Initialization can be done using **initialization lists**:

```
//for single struct
detector_t DE = {1000., 0., 17, 3};
//for nested structs
telescope_t telescope = {{1000., 0., 17, 3},{650., 0., 9, 4},2};
```

NOTE

- It is a good practice to define struct in a header file, so that it is accessible for multiple files
- struct can be also used as an external variable
- Understanding struct is a first step to understanding object oriented programming!

Boolean values

Declaration and assignment

```
bool bValue1 = true; // explicit assignment
bool bValue2(false); // implicit assignment
bool bValue3 = !true; // bValue3 will have the value false
bool bValue4(!false); // bValue4 will have the value true

cout << bValue1 << " " << bValue2 << end;
```

What will cout in the above example print?

```
#include <iostream>
using namespace std;

// returns true if x=y
bool IsEqual(int x, int y)
{
    return (x == y); // equality operator
}
int main()
{
    cout << "Enter a value: ";
    int x;
    cin >> x;

    cout << "Enter another value: ";
    int y;
    cin >> y;

    if (IsEqual(x, y))
        cout << "equal"<<endl;
    else
        cout << "not equal"<<endl;
    return 0;
}
```

Char

```
char chValue = 'a';
char chValue2 = 97; //assign an ASCII code

cout << chChar;
cout << (int)chChar;
```

NOTE

What will happen if user enters multiple characters to `cin`?

```
#include "iostream";

int main()
{
    using namespace std;
    char chChar;
    cout << "Input a keyboard character: ";
    cin >> chChar;
    cout << chChar << " has ASCII code " << (int)chChar << endl;
}
```

Operators

Arithmetic operators

operator	description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Compound assignments

expression	equivalent
y += x;	y = y + x;
x -= 5;	x = x - 5;
x /= y;	x = x / y;
y *= x + 1;	y = y * (x+1);

Increment and decrement

These are equivalent statements:

```
++x;
x+=1;
x=x+1;
```

NOTE

```
int x = 3;
int y, z;
y = x++;
z = ++x;
```

What values of y and z will this code output?

Operators

Relational operators

Operator	Form
Greater than	x > y
Less than	x < y
Greater than or equals	x >= y
Less than or equals	x <= y
Equality	x == y
Inequality	x != y

All the operators above return a boolean value.

Logical operators

Operator	Form
logical NOT	!x
logical AND	x && y
logical OR	x || y

De Morgan's law

`!(x || y)`	`!x && !y`
`!(x && y)`	`!x || !y`

Mathematical functions in C++

Header `#include <math.h>` contains a set of functions to compute common mathematical operations and transformations. Among the available functions are:

`cos(x)`	Compute cosine
`sin(x)`	Compute sine
`tan(x)`	Compute tangent
`acos(x)`	Compute arc cosine
`asin(x)`	Compute arc sine
`atan(x)`	Compute arc tangent
`exp(x)`	Compute exponential function
`log(x)`	Compute natural logarithm
`log10(x)`	Compute common logarithm

`pow(x,y)`	Raise x to power of y
`sqrt(x)`	Compute square root
`cbrt(x)`	Compute cubic root
`ceil(x)`	Round up value
`floor(x)`	Round down value
`fmod(x,y)`	Compute remainder of division x/y
`trunc(x)`	Truncate value (round toward zero, to the nearest integral value)
`round(x)`	Round to nearest integral value
`abs(x)`	Compute absolute value