

“Simple” C++ Programs

The Nitty-Gritty

July 1, 2010

Computation Headaches

Integer Computations

Type Specifier	Positive Range
int	0 - 2,147,483,647

```
1 // note the absence of commas in the
2 // numbers
3 int bigInt1 = 2000000000;
4 int bigInt2 = 1111111111;
5
6 cout << bigInt1 << "+" <<
7      bigInt2 << "=" <<
8      bigInt1 + bigInt2 << endl;
```

Integer Computations

Type Specifier	Positive Range
int	0 - 2,147,483,647

Overflow

```
1 // note the absence of commas in the
2 // numbers
3 int bigInt1 = 2000000000;
4 int bigInt2 = 1111111111;
5
6 cout << bigInt1 << "+" <<
7     bigInt2 << "=" <<
8     bigInt1 + bigInt2 << endl;
```

```
2000000000 + 1111111111 = -1183856185
```

RUN

EDIT

int_overflow.cxx

Integer Computations

Type Specifier	Positive Range
int	0 - 2,147,483,647

Truncated Division

```
1 // initialize two variables to
2 // one and a half
3 int integerHalf( 3 / 2 );
4 double doubleHalf( 3 / 2 );
5
6 cout.setf( ios::fixed );
7 cout.precision(4);
8 cout << "integerHalf_" <<
9     integerHalf << endl;
10
11 cout << "doubleHalf_" <<
12     doubleHalf << endl;
```

```
integerHalf 1
doubleHalf 1.0000
```

RUN

EDIT

int_truncated.cxx

Integer Computations

Type Specifier	Positive Range
int	0 - 2,147,483,647

“Decimal Casting & Promotion”

```
1 // initialize a variable to
2 // one and a half using decimal casting
3 double doubleHalf( 3.0 / 2 );
4
5 cout.setf( ios::fixed );
6 cout.precision(4);
7
8 cout << "doubleHalf_ " <<
9     doubleHalf << endl;
```

```
doubleHalf 1.5000
RUN          EDIT          decimal_casting.cxx
```

ℝ Underflow & Overflow

Type Specifier	Positive Range
double	1.7977×10^{-308} - 2.2251×10^{308}

Underflow: You cannot compute $\frac{1.0 \times 10^{-308}}{10} = 1 \times 10^{-309}$

Overflow: You cannot compute $10(2 \times 10^{308}) = 2 \times 10^{309}$

Initial Values of Variables

Initial Values of Fundamental Types

Fundamental types (everything except string) have random values if they are not initialized when declared.

```
1 int Integer;  
2 double Double;  
3 string String;  
4  
5 cout << "Integer=" << Integer << endl;  
6 cout << "Double=" << Double << endl;  
7 cout << "String='" << String << "' << endl;
```

```
Integer=-1860988492  
Double=8.69475e-311  
String=''
```

RUN

EDIT

garbage_variables.cxx

Variable Names ~ Identifiers

The char Variable Type

char type

Think of a char as an integer type that cin and cout treat specially.

It still just holds a small integer value!

```
1 char ch('x');
2 int ch_value;
3
4 cout << ch ;
5 ch = ch + 1;
6 cout << ch ;
7 ch = ch + 1;
8 cout << ch ;
9 ch = ch + 1;
10 cout << ch << endl;
11
12 cout << "Type a single character:_" << flush;
13 cin >> ch;
14 ch_value = ch;
15 cout << endl << "Thanks ,you typed:_" << ch
16     << " '_=" << ch_value << endl;
```

<<Interactive Program>>

RUN

EDIT

char_type.cxx

ASCII Codes for Characters

	30	40	50	60	70	80	90	100	110	120
0		(2	<	F	P	Z	d	n	x
1)	3	=	G	Q	[e	o	y
2		*	4	>	H	R	\	f	p	z
3	!	+	5	?	I	S]	g	q	{
4	"	,	6	@	J	T	^	h	r	
5	#	-	7	A	K	U	_	i	s	}
6	\$.	8	B	L	V	'	j	t	~
7	%	/	9	C	M	W	a	k	u	
8	&	0	:	D	N	X	b	l	v	
9	'	1	;	E	O	Y	c	m	w	

Newlines and Tabs

You should know about three special “escape sequences”:

Newline When `'\n'` is sent to the console, the cursor wraps to the next line (subtly different from `endl`).

Tab `'\t'` Jumps cursor ahead to an “eighth multiple” console column.

Backslash (a.k.a. Whack) Use `'\\'` to have a single `'\'` printed to the console.

```
1 char n = '\n';
2 char t = '\t';
3
4 // print column numbers
5 cout << "012345678901234567890123456789 ";
6 cout << "01234567890123456789 ";
7 cout << endl;
8 cout << '\n'; // creates an empty line
9
10 cout << "These" << '\t' << "words" << t;
11 cout << "are" << t << "tabbed" << n;
12 cout << "These\twords\tare\ttabbed\n" ;
```

```
01234567890123456789012345678901234567890123456789
```

```
These words are tabbed
```

```
These words are tabbed
```

RUN

EDIT

newlines_and_tabs.cxx

Variables in C++

Beware the Reserved Words!

asm	auto	break	case	catch	char
class	const	continue	default	delete	do
double	else	enum	extern	float	for
friend	goto	if	inline	int	long
new	operator	private	protected	public	register
return	short	signed	sizeof	static	struct
switch	template	this	throw	try	typedef
union	unsigned	virtual	void	volatile	while

Fundamental Variable Types

Values for Intel PC 32b Architectures

Type Specifier	Bytes (Bits)	Negative Range		Positive Range	
bool	1 (1)			0	1
char	1 (8)	-128	0	0	127
unsigned char	1 (8)			0	255
short	2 (16)	-32,768	0	0	32,767
unsigned short	2 (16)			0	65,535
int	4 (32)	-2,147,483,648	0	0	2,147,483,647
unsigned int	4 (32)			0	4,294,967,295
float	8 (64)	-3.4028×10^{38}	-1.1755×10^{-38}	1.1755×10^{-38}	3.4028×10^{38}
double	16 (128)	-2.2251×10^{308}	-1.7977×10^{-308}	1.7977×10^{-308}	2.2251×10^{308}

floats and doubles may also be zero.

floats have 6 digits of decimal precision (6 significant digits)

doubles have 15 digits of decimal precision (15 significant digits)

On a 64b machine, a signed int ranges through
 $[-9223372036854775808, 9223372036854775807]$

Answer worksheet questions 6 and 7.

Operators in C++

Operators

arithmetic operator An operator is a (reserved) symbol in C++ that modifies or creates a result from one or more “nearby” symbols.

Type	Example
+ is a Binary Operator	3 + 4
- is a Binary Operator	3 - 4
-negation is a Unary Operator	-3

Precedence and Associativity

Every operator in C++ has *precedence* and *associativity*.

Precedence Without ()'s, the order in which operators are evaluated.

Associativity Of the other symbols around the operator, which will it operate on?

Arithmetic Operators

Modulus Operator (%)

```
1 int answer;  
2  
3 answer = 3 % 4;  
4 cout << "3 % 4 = " << answer << endl;  
5 answer = 5 % 4;  
6 cout << "5 % 4 = " << answer << endl;  
7 answer = 64 % 4;  
8 cout << "64 % 4 = " << answer << endl;  
9 answer = 125 % 13;  
10 cout << "125 % 13 = " << answer << endl;
```

```
3 % 4 = 3  
5 % 4 = 1  
64 % 4 = 0  
125 % 13 = 8
```

RUN

EDIT

modulus_op.cxx

Modulus Operator (%)

A binary operator that returns the *remainder* of the left argument divided by the right argument.

Valid for only non-negative integer arguments, the right argument may not be 0.

Left-to-Right associativity, **precedence equal to multiplication and division.**

```
1 int answer;  
2  
3 answer = 3 % 4;  
4 cout << "3 % 4 = " << answer << endl;  
5 answer = 5 % 4;  
6 cout << "5 % 4 = " << answer << endl;  
7 answer = 64 % 4;  
8 cout << "64 % 4 = " << answer << endl;  
9 answer = 125 % 13;  
10 cout << "125 % 13 = " << answer << endl;
```

```
3 % 4 = 3  
5 % 4 = 1  
64 % 4 = 0  
125 % 13 = 8
```

RUN

EDIT

modulus_op.cxx

Assignment Operators

= += -= *= /= %= & Multiple Assignment

```

1 int x(0), y(1), z(2);
2
3 x = x + 1;
4 // abbreviated assignment
5 x += z;
6 cout << "x=" << x << " y=" << y
7     << " z=" << z << endl;
8
9 // multiple assignment. R-to-L
10 // means everyone gets x's value!
11 y = z = x;
12 cout << "x=" << x << " y=" << y
13     << " z=" << z << endl;
14
15 z -= 1;
16 y %= z;
17 x *= z;
18 cout << "x=" << x << " y=" << y
19     << " z=" << z << endl;

```

x=3	y=1	z=2
x=3	y=3	z=3
x=6	y=1	z=2

RUN

EDIT

assignment_ops.cpp

Assignment Operators

$$x \ ?= \ y; \quad \iff \quad x = x \ ? \ y;$$

Assignment operators have the “highest” precedence, so **evaluated last**.

Right-to-Left associativity.

Try worksheet questions 10, 11, and 12.

Increment and Decrement Operators

Two different notations:

Prefix Increment:	<code>++x;</code>	\iff	<code>x = x + 1;</code>
Prefix Decrement:	<code>--x;</code>	\iff	<code>x = x - 1;</code>
Postfix Increment:	<code>x++;</code>	\iff	<code>x = x + 1;</code>
Postfix Decrement:	<code>x--;</code>	\iff	<code>x = x - 1;</code>

Both prefix and postfix are unary operators, and synonymous with the statements on the right hand side **when they are alone in a C++ statement.**

Increment and Decrement Operators

Prefix Increment

```
1 int x(0), y(1), z(2);
2 cout << "x=" << x << " "
3     << "y=" << y << " "
4     << "z=" << z << endl;
5
6 x = y + ++z;
7 cout << "x=" << x << " "
8     << "y=" << y << " "
9     << "z=" << z << endl;
```

x=0 y=1 z=2

x=4 y=1 z=3

RUN

EDIT

prefix_inc.cxx

Increment and Decrement Operators

Prefix Increment

```

1 int x(0), y(1), z(2);
2 cout << "x=" << x << " "
3     << "y=" << y << " "
4     << "z=" << z << endl;
5
6 x = y + ++z;
7 cout << "x=" << x << " "
8     << "y=" << y << " "
9     << "z=" << z << endl;

```

```

x=0 y=1 z=2
x=4 y=1 z=3

```

RUN

EDIT

prefix_inc.cxx

Postfix Increment

```

1 int x(0), y(1), z(2);
2 cout << "x=" << x << " "
3     << "y=" << y << " "
4     << "z=" << z << endl;
5
6 x = y + z++;
7 cout << "x=" << x << " "
8     << "y=" << y << " "
9     << "z=" << z << endl;

```

```

x=0 y=1 z=2
x=3 y=1 z=3

```

RUN

EDIT

postfix_inc.cxx

Increment and Decrement Operators

Prefix operators change the variable **before** it is used in computations.

Postfix operators change the variable **after** it is used in computations.

Increment and Decrement Operators

Don't use them if the variable is referenced more than once in a statement, since it easy to fall into *undefined behavior*:

```
x = y++ + z-- + y - z;
```

which will generate **different results from different compilers**.

Using them in multiple assignment statements is bad form, they are simply hard to read.

If you're concerned about the validity of a computation with prefix and postfix operators, then break it up into several smaller statements.

You can only use these operators on variables, **3++ will not compile**.

Try the first two expressions in question 13.

Non-Arithmetic Operators

Member Accessor

The `.` is called the “member-dot-operator” in C++.

Objects in C++ expose functions to the programmer using the dot-operator.

`cout` and `cin` are two C++ objects we've already seen in action.

Dot-Operator Example: cout Formatting

- ▶ `#include <iomanip>`
- ▶ `cout.setf()` controls the output “mode.”
- ▶ Use `cout.precision()` or `setprecision()` to control the number of decimal digits printed (called the *precision* in C++).

```
1 cout.setf( ios::fixed);
2 cout.precision(1);
3 cout << 0.12345 << endl;
4
5 cout.precision(2);
6 cout << 0.12345 << endl;
7
8 cout.precision(4);
9 cout << 0.12345 << endl;
10
11 cout.precision(3);
12 cout << 0.12345 << endl;
```

```
0.1
0.12
0.1235
0.123
RUN EDIT ios_example.cxx
```

Cast Operator

“Decimal Casting”

```
1 // initialize a variable to  
2 // one and a half using decimal casting  
3 double doubleHalf( 3.0 / 2 );
```

Cast Operator

We can also use an explicit
cast operator:
`typename(variable)`

“Decimal Casting”

```
1 // initialize a variable to  
2 // one and a half using decimal casting  
3 double doubleHalf( 3.0 / 2 );
```

A Right-to-Left Unary Operator

```
1 double doubleHalf( double(3)/2 );  
2 int intThree(3), intTwo(2);  
3 double secondHalf( intThree/double(intTwo) );  
4  
5 cout.setf( ios::fixed );  
6 cout.precision(4);  
7  
8 cout << "doubleHalf_ " <<  
9     doubleHalf << endl;  
10 cout << "secondHalf_ " <<  
11     secondHalf << endl;
```

Cast Operator

We could have used this yesterday...

```
1 char ch;  
2  
3 cout << "Type a single character:" << flush;  
4 cin >> ch;  
5 cout << endl << "Thanks, you typed:" << ch  
6   << "' = " << int(ch) << endl;
```

<<Interactive Program>>

RUN

EDIT

char_type_casted.cxx

The Precedence Table

Precedence	Operator(s)	Associativity	Notes
First	()	innermost	
:	Unary: ++ -- .	⇒	Postfix++
:	Unary: ++ -- + - cast()	⇐	++Prefix
:	Binary: * / %	⇒	
:	Binary: + -	⇒	
Last	Assignment: = += -= *= /= %=	⇐	

finis