Introduction to C++

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Dr. Richard Berger
richard.berger@temple.edu

Department of Mathematics
Temple University

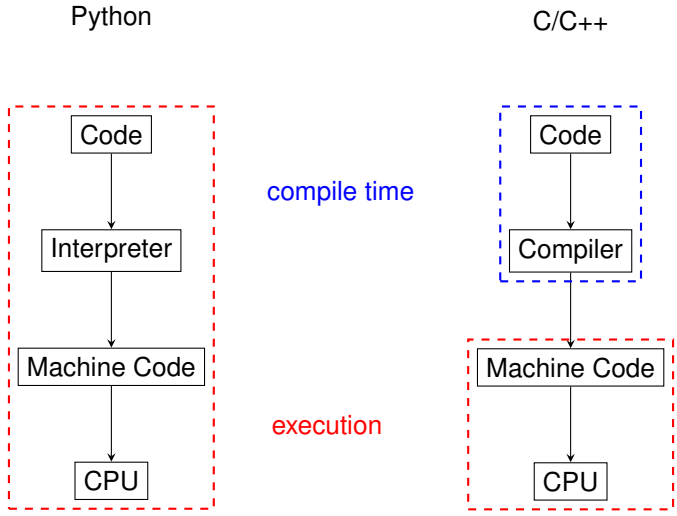
10/11/2016

TEMPLE UNIVERSITY

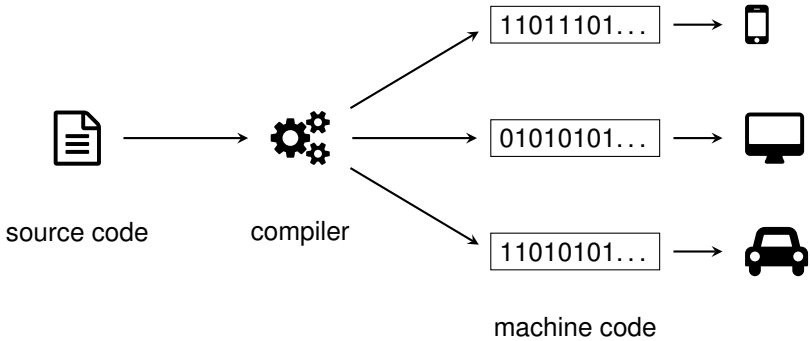
Outline

Outline

Interpreted vs. Compiled Languages

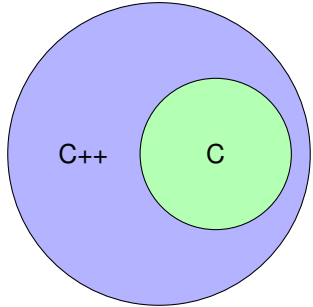


Portable Code



The C Programming Language

- Popular systems programming language
- Implementation language of the UNIX operating systems
- Very portable and efficient
- C++ is a superset of C



What is C++?

C++ is one of the most widely used programming languages today. Many software products you use everyday are programmed in C/C++.

Popular programs implemented in C++

- Microsoft Windows
- Microsoft Office
- Adobe Photoshop
- Facebook uses C++ on the backend
- All major browsers (Firefox, Chrome, Safari)
- 3D Game Engines (Unreal Engine, Cry Engine)

TIOBE Index

Table: Most popular programming languages (September 2016)

1	Java	18.236%
2	C	10.955%
3	C++	6.657%
4	C#	5.493%
5	Python	4.302%
6	JavaScript	2.929%
7	PHP	2.847%
8	Assembly language	2.417%
9	Visual Basic .NET	2.343%
10	Perl	2.333%

History

1969-1973	Development of the C Programming Language
1979	Bjarne Stroustrup started developing *"C with Classes"*
1984	*"C with Classes"* is renamed to C++
1998	First ISO C++ standard was ratified (C++98)
2001	Bugfix release of the standard (C++2001)
2011	Second ISO C++ standard (C++11) ratified. Previously known as C++0x (they were optimistic)
2014	C++14 standard
2017	Upcoming C++17 standard

ISO C++ Standard

This standard consists of two parts:

Core language

- Syntax Rules
- Keywords
- Statements
- Operators

Standard library

- General utilities
- Strings
- Data Structures
- Algorithms
- Numerics
- Input/output

Comparison to Python

Python

- architecture independent
- “batteries included”
- rich standard library
- rich ecosystem (pip)
- automatic memory management / garbage collection
- good for general purpose applications, but for expensive computations most modules fall back to “native” code (which is usually written in C/C++)

C/C++

- architecture dependent
- harder to write
- general purpose library
- starting with C++11 standard library is rapidly expanding
- libraries easily extend code, but are not as convenient and error prone in their usage
- manual memory management
- easier to write efficient and fast code

C++ Learning Resources

Online Resources

- http://cppreference.com
- http://mindview.net/Books/TICPP/ThinkingInCPP2e.html

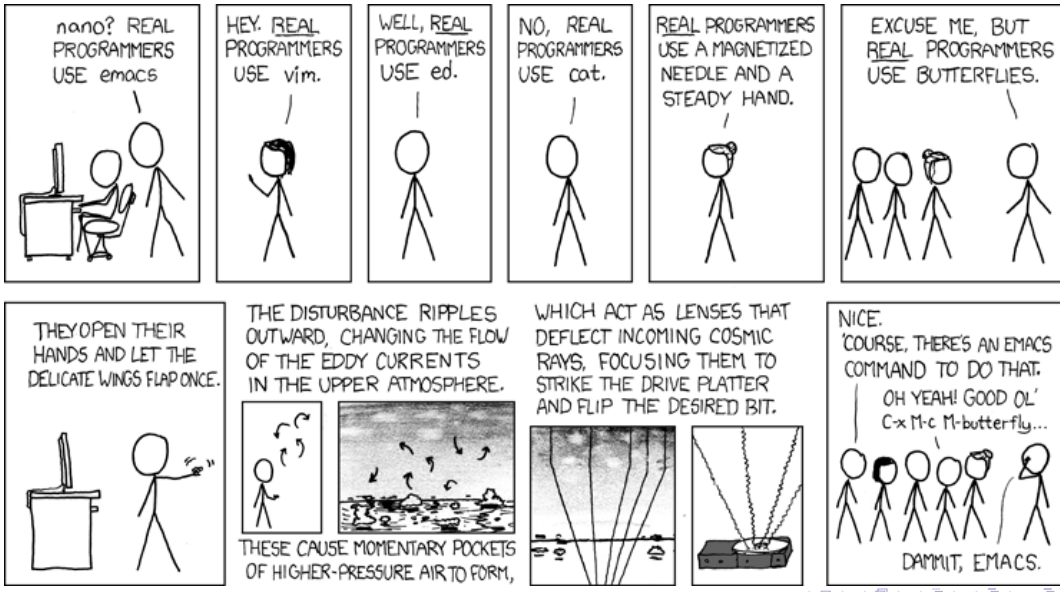
Books

- The C++ Programming Language, 4th Edition
- The C++ Standard Library: Tutorial and Reference
- The C Programming Language

What you **need** to program in C++

1. a text editor
   - vim
   - emacs
   - gedit
   - Notepad++
2. a C++ compiler
   - GNU Compiler Collection (GCC)
   - Clang
   - Intel Compiler
   - Microsoft Visual C++

“Real Programmers”



Know your text editor and know it well

Practise makes perfect

- A tennis player will perfect using his/her tennis racket
- A music player will perfect using his/her instrument
- A artist will learn about his/her brushes and materials
- **A programmer's #1 tool is his/her text editor**

vim

Create a configuration file for vim by putting a hidden file in your home directory:

```
vim ~/.vimrc
```

Good defaults are the following contents:

```
syntax on          # enables syntax highlighting
set tabstop=4      # makes tab stops 4 spaces wide
set shiftwidth=4   # shifts 4 spaces for indent
set expandtab      # expands tabs to spaces
```

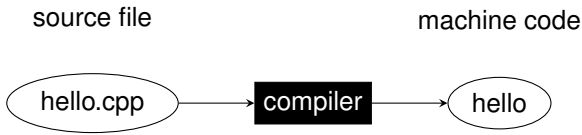
Outline

Your first C++ program

Following the tradition in most programming languages, we are going to write our first C++ program that will print out "Hello World!" to the console.

```
int main() {
    puts("Hello World!");
    return 0;
}
```

Compilation



- a compiler translates C++ code (text) into machine code (binary)
- each source file corresponds to one translation unit

Compilation using the GNU C++ compiler

Before compilation

```
ls -la
drwxr-xr-x 2 richard richard 4096 Jun 24 12:12 .
drwxr-xr-x 4 richard richard 4096 Jun 24 12:01 ..
-rw-r--r-- 1 richard richard  104 Jun 24 12:01 hello.cpp
```

After compilation

```
g++ -o hello hello.cpp
```

```
ls -la
drwxr-xr-x 2 richard richard 4096 Jun 24 12:12 .
drwxr-xr-x 4 richard richard 4096 Jun 24 12:01 ..
-rwxr-xr-x 1 richard richard 8488 Jun 24 12:13 hello
-rw-r--r-- 1 richard richard  104 Jun 24 12:01 hello.cpp
```

Compilation using the GNU C++ compiler

Before compilation

```
ls -la
drwxr-xr-x 2 richard richard 4096 Jun 24 12:12 .
drwxr-xr-x 4 richard richard 4096 Jun 24 12:01 ..
-rw-r--r-- 1 richard richard  104 Jun 24 12:01 hello.cpp
```

After compilation

```
g++ -o hello hello.cpp
```

```
ls -la
drwxr-xr-x 2 richard richard 4096 Jun 24 12:12 .
drwxr-xr-x 4 richard richard 4096 Jun 24 12:01 ..
-rwxr-xr-x 1 richard richard 8488 Jun 24 12:13 hello
-rw-r--r-- 1 richard richard  104 Jun 24 12:01 hello.cpp
```

Compilation using the GNU C++ compiler

Before compilation

```
ls -la
drwxr-xr-x 2 richard richard 4096 Jun 24 12:12 .
drwxr-xr-x 4 richard richard 4096 Jun 24 12:01 ..
-rw-r--r-- 1 richard richard  104 Jun 24 12:01 hello.cpp
```

After compilation

```
g++ -o hello hello.cpp
```

```
ls -la
drwxr-xr-x 2 richard richard 4096 Jun 24 12:12 .
drwxr-xr-x 4 richard richard 4096 Jun 24 12:01 ..
-rwxr-xr-x 1 richard richard 8488 Jun 24 12:13 hello
-rw-r--r-- 1 richard richard  104 Jun 24 12:01 hello.cpp
```

Live Demo

Your first C++ compile error

```
g++ -o hello hello.cpp
hello.cpp: In function 'int main()':
hello.cpp:2:24: error: 'puts' was not declared in this scope
     puts("Hello World!");
                        ^
```

```
int main() {
    puts("Hello World!");
    return 0;
}
```

Compilation

- During compilation the compiler will NOT run your code
- The compiler only reads the C++ code, line by line, word for word
- Everything used must be known to the compiler
- If you introduce new names in your program, you must tell the compiler *what* it is

`puts` example:

We have to tell the compiler what `puts` is. We want to use the `puts` function from the C standard library, defined by the POSIX standard.

Using functions from system libraries

- functions such as `puts` are declared in header files
- you have to *include* the correct header in your program in order to use its functions
- if a function is part of the POSIX standard, use `man [FUNCTION]` to learn about which header is needed.

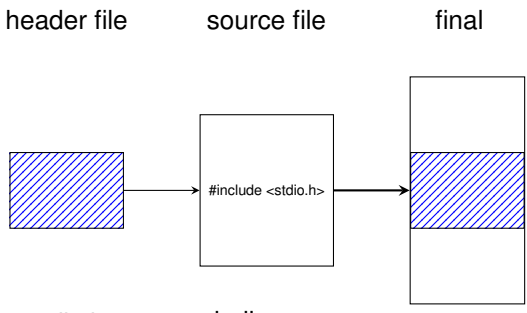
```
#include <stdio.h>

int main() {
    puts("Hello World!");
    return 0;
}
```

- headers are just another form of C++ code files
- their name usally ends with .h, but you will also find .hh and .hpp

File inclusion

- the **`#include`**-directive copies the contents of a given header into your source code during compilation
- so when the compiler processes the final file, it will first see the contents of `stdio.h` and *learn* about `puts`
- afterwards `puts` can be used because it is known to the compiler



Live Demo

Outline

Basic structure

```
int main() {
    // this is a comment
    statement1;
    statement2;
    statement3; // this too

    /* this is
       a multi line
       comment */

    return 0;
}
```

- Every C++ program starts in the `main`-function
- `main` contains the code which is executed
- Each statement ends with a semicolon ;
- The number returned by the main function is its exit code. By convention, 0 signifies that everything was ok. Non-zero values indicate that an error has occured.

Simple Expressions

*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulus (remainder after division)

Parentheses may be used to group terms.

```
3 * (4 + 4)
3 + 8 * 4 + 4
3 + (8 * 4) + 4
```

Using Expressions

```
int main() {
    3 + (8 * 4) + 4;
    return 0;
}
```

While this is a valid program, it doesn't do anything with the result. We want to store the result of the computation and use it somewhere else.

Dynamic typing vs statically typing

Python

```
a = 10
# a is an int

a = 10.0
# now it is a float

a = "text"
# now it is a string
```

The data type of variables can change at any time and is determined dynamically.

C/C++

```
int a = 10;
// a is an int

float b = 10.0f;
// b is a float

char * s = "text";
// s is a C-String
```

Variables are declared of *one* data type which can not change.

Declaration

- Introduces a *name* to the compiler and specifies its *data type*
- A variable can only be declared once inside a scope
- Variables must be declared before being used

```
int variableName;
```

data type

name

Definition

Copies the value on the right side to the named variable on the left

```
variableName = VALUE;
```

Both can be combined into one statement:

```
int variableName = 42;
```

Outline

Contents of a Computer Program

- the smallest unit of information is one **bit**, which is either 0 or 1
- any information on a computer is a sequence of bits
- a program consists of two things: data and instructions

data Information needed by the program to run

instructions Sequence of commands which the processor should execute. This is machine code dependent on the processor.

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

with 2 bit, you can store 4 values

```
00, 01, 10, 11
```

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

with 2 bit, you can store 4 values

```
00, 01, 10, 11
```

with 3 bit, you can store 8 values

```
000, 001, 010, 011, 100, 101, 110, 111
```

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

with 2 bit, you can store 4 values

```
00, 01, 10, 11
```

with 3 bit, you can store 8 values

```
000, 001, 010, 011, 100, 101, 110, 111
```

with 4 bit, you can store 16 values

```
0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
```

Storing information in binary

$$\text{number of values represented by } b \text{ bits} = 2^b$$

- 1 byte is 8 bits $\Rightarrow 2^8 = 256$
- 2 bytes are 16 bits $\Rightarrow 2^{16} = 65,536$
- 4 bytes are 32 bits $\Rightarrow 2^{32} = 4,294,967,296$
- 8 bytes are 64 bits $\Rightarrow 2^{64} = 18,446,744,073,709,551,616$

Fundamental Data Types

Processors have two different modes of doing calculations:

- integer arithmetic
- floating-point arithmetic

The operands of these calculations have to be stored in binary form. Because of this there two groups of fundamental data types for numbers in a computer:

- integer data types
- floating-point data types

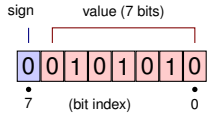
Integer Types[1]

char	8 bit (1 byte)
short int	16 bit (2 byte)
int	32 bit (4 bytes)
long int	64 bit (8 bytes)

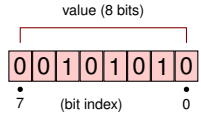
- **short int** can be abbreviated as **short**
- **long int** can be abbreviated as **long**
- integers are by default **signed**, and can be made **unsigned**

[1] sizes only valid on Linux x86_64

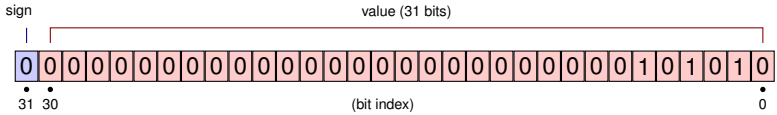
char



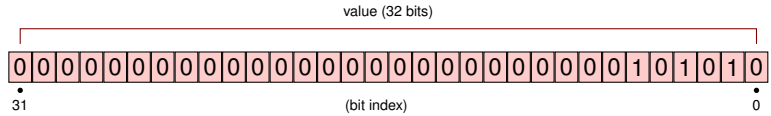
unsigned char



int



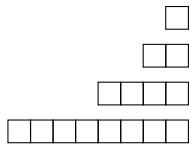
unsigned int



Integer Types - Value Ranges

`char`	$[-128, 127]$
`short`	$[-32768, 32767]$
`int`	$[-2147483648, 2147483647]$
`long`	$[-9223372036854775808, 9223372036854775807]$
`signed char`	$[-128, 127]$
`signed short`	$[-32768, 32767]$
`signed int`	$[-2147483648, 2147483647]$
`signed long`	$[-9223372036854775808, 9223372036854775807]$
`unsigned char`	$[0, 255]$
`unsigned short`	$[0, 65535]$
`unsigned int`	$[0, 4294967295]$
`unsigned long`	$[0, 18446744073709551615]$

Type Compatibility



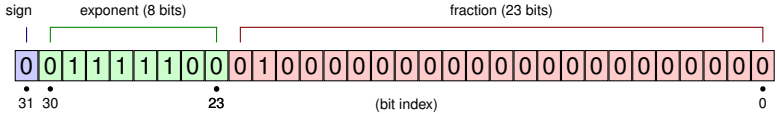
- **`char`** < **`short`** < **`int`** < **`long`**
- in words: you can store values of a **`char`** in a **`short`**. Both of them can be stored in an **`int`**. Finally **`long`** has the largest value range and can store all three smaller types.
- the similar logic applies for **`unsigned`** types

Floating-Point Types

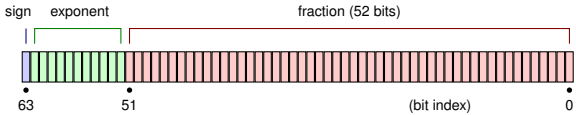
`float`	32 bit (4 bytes)	single precision	$\approx$ 7 decimal digits
`double`	64 bit (8 bytes)	double precision	$\approx$ 15 decimal digits

- **`float`** numbers are between 10^{-38} to 10^{38}
- **`double`** numbers are between 10^{-308} to 10^{308}

float



double



`sizeof` operator

- returns the number of bytes a data type occupies in memory

`sizeof(char)`	1
`sizeof(short)`	2
`sizeof(int)`	4
`sizeof(long)`	8
`sizeof(float)`	4
`sizeof(double)`	8

Table: Data type sizes in Linux with GCC compiler on x86_64

- it can also return the size of a literal

Live Demo

Integer Literals

- a *literal* represents a fixed value in code: e.g., `42`
- there multiple ways to write integer literals in C++
- if you write just a number, the compiler will choose the smallest data type it will fit into from the following list: **`int`**, **`long`**, **`long long`**.

Example:

```
sizeof(42)          // = 4 (int)
sizeof(4294967295)  // = 8 (long)
```

Integer Literals

- The data type of the literal can be changed using suffixes
- Appending `u` or `U` to an integer makes it an **`unsigned`** type
- Appending `l` or `L` to an integer makes it a **`long`** type
- Appending `ul` or `UL` makes it a **`unsigned long`** type

Example:

```
sizeof(42u)            // = 4 (unsigned int)
sizeof(42ul)           // = 8 (unsigned long)
sizeof(4294967295U)    // = 8 (unsigned long)
```

Floating-Point Literals

- by default a floating-point literal is a **`double`**
- appending `f` or `F` to a floating-point literal makes it a **`float`**
- leading zero is optional
- you can use scientific notation using either `e` or `E`

Examples:

```
sizeof(42.0)   // = 8 (double)
sizeof(42.0f)  // = 4 (float)
.5             // short form
42.0e10        // scientific notation
6.02214086E23
1.38064852e-23
```

Characters and Character Literals

- **char** not only represents 8 bit numbers, but is mainly used for representing text characters.
- each text character in ASCII is mapped to a 8 bit integer number.
- in other words: a character is simply an integer number *interpreted* as a text character
- a character literal is defined using single quotes

Example:

```
char a = 'A';    // equal to 65
char b = 'B';    // equal to 66
char c = b + 1; // equal to 67 which is 'C'
char d = 'a';    // equal to 97
char e = 'b';    // equal to 98
char f = 'B' + ('a'-'A');
```

Special Characters

`\n`	newline
`\t`	tabulator
`\r`	catridge return
`\0`	NUL
`\"`	Escaped double quote
`\\`	Escaped backslash

C-Strings

- sequences of characters are called strings
- a string literal is defined using double quotes
- they are often called C-Strings because these are the most primitive form of text strings which existed since C

```
printf("some text ");
printf("that will be in the same line\n");
printf("some other text on its own line\n");
```

Output

```
some text that will be in the same line
some other text on its own line
```

Booleans

- Can only have one of two values: **`true`** and **`false`**
- C didn't have Booleans. It used integers for storing such information. This is why they are compatible. You can use any integer result as Boolean expression.
- The integer value 0 is associated to **`false`**, while anything else is considered **`true`**

```
bool b1 = true;
bool b2 = false;
bool b3 = 2; // true
bool b4 = 0; // false
bool is_odd = 5 % 2; // = 1 => true
```

Outline

Basic Output

- `puts` automatically starts a new line after its output
- `printf` needs the `'\n'` character at the end to start a new line

```
puts("Hello World!");
```

```
printf("Hello World!\n");
```

Formatted Output with `printf`

- `printf` takes one or more arguments
- defined in the `stdio.h` header
- the first argument is a string which may contain placeholders

Placeholders

%d	decimal number
%ud	unsigned decimal number
%f	floating-point number
%s	C-string
%x	decimal as hexadecimal
%p	pointer (next week)

printf examples

```
int a = 10;
printf("the value is %d\n", a);
```

```
the value is 10
```

printf examples

```
double x = 10.0;
double y = 20.0;
printf("Point =(%f, %f)\n", x, y);
```

```
Point = (10.0, 20.0)
```

Getting user input using `scanf`

- `scanf` is the inverse of `printf`
- instead of printing a formatting string with placeholders to a screen, it accepts user input in that format.
- the placeholders are used to save the entered data into variables
- variables must be given to the function using their address: `&var`

```
int value = 0;
printf("Enter a value: ")
scanf("%d", &value);
printf("The value was %d!\n", value);
```

```
Enter a value:
42
The value was 42!
```

Putting it all together

```
#include <cstdio>

int main() {
    int a = 0;
    int b = 0;

    puts("Enter first number:");
    scanf("%d", &a);

    puts("Enter second number:");
    scanf("%d", &b);

    int sum = a + b;
    printf("The sum is %d\n", sum);

    return 0;
}
```

Live Demo

Outline

Relational Operators

Operator	Meaning
<=	Less than or equal to
<	Less than
>	Greater than
>=	Greater than or equal to
==	Equal
!=	Not equal

Examples:

```
int a = 3;
int b = 5;
bool c = a < 4;     // true
bool d = b > 6;     // false
bool e = (b == 5); // true
```

Logical Operators

Operator	Name	Usage
||	logical OR	(expr1) || (expr2)
&&	logical AND	(expr1) && (expr2)
!	logical NOT	!(expr1)

Examples:

```
int a = 3;
int b = 5;
bool c = (a == 3) || (b == 5); // true
bool d = (a > 1) && (a < 5);   // true
bool e = !true;                // false
```

Short-circuit evaluation

Logical OR

```
exprA || exprB || exprC
```

Evaluation stops after one expression is **`true`**

Logical AND

```
exprA && exprB && exprC
```

Evaluation stops after one expression is **`false`**

- expressions are evaluated left to right, from inside out, stopping as soon as the result can no longer change.

implicit type conversion

- arithmetic has to be performed using the same data type
- if you mix data types in an expression, the compiler will attempt to implicitly convert towards the larger data type

Type promotion

```
int a = 3;
double b = 3.3;

printf("%f", a+b); // implicit double+double
printf("sizeof(a+b) = ", sizeof(a+b));
```

```
6.3
sizeof(a+b) = 8
```

explicit type conversion

- the compiler will only do conversions to larger types automatically
- for smaller types you have to explicitly tell the compiler what you want by *casting* an expression to a specific data type.
- final result depends on when a cast happens

```
double a = 3.1;
double b = 3.3;
double c = 3.6;

int result = (int)a + (int)b + (int)c; // = 9

int result2 = (int)(a + b + c);        // = 10
```

Division

Floating-Point

```
double a = 1.0 / 3.0;
double b = 10.0 / 3.0;

// a = 0.33333...
// b = 3.33333...
```

Integer

```
int a = 1 / 3;
int b = 10 / 3;

// a = 0
// b = 3
```

Shorthands for arithmetic

- operators accessing and changing the same variable can be written in a more compact notation

long form	short form
`a = a + x`	`a += x`
`a = a - x`	`a -= x`
`a = a * x`	`a *= x`
`a = a / x`	`a /= x`
`a = a % x`	`a %= x`

Increment operator

- incrementing an integer number by one can be done using the increment ++ operator
- depending on whether the operator is before or after a variable name determines when the operation takes place
- the prefix versions immediately change the value of the given variable and returns the updated value
- the postfix version first returns the current value and then updates

Pre-increment

```
int i = 1;
a = ++i;   // a = 2
b = i;     // b = 2
```

Post-increment

```
int i = 1;
a = i++;   // a = 1
b = i;     // b = 2
```

Decrement operator

- decrementing an integer number by one can be done using the decrement $--$ operator
- depending on whether the operator is before or after a variable name determines when the operation takes place
- the prefix versions immediately change the value of the given variable and returns the updated value
- the postfix version first returns the current value and then updates

Pre-decrement

```
int i = 1;
a = --i;   // a = 0
b = i;     // b = 0
```

Post-decrement

```
int i = 1;
a = i--;   // a = 1
b = i;     // b = 0
```

Integer Literal - Number Formats

- Integer literals come in 3 forms: decimal, octal and hexadecimal

decimal: `2016`

octal: `03740` (number start with leading `0`)

hexadecimal: `0x7E0` (number start with leading `0x`)

- the reason for octal and hexadecimal are useful for programmer is because they both have a basis which is a power of 2
- this makes it easy to create binary numbers using either of these two notations

Digits of various number representations

Decimal (Base 10)

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Binary (Base 2)

0, 1

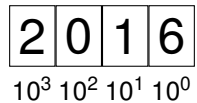
Octal (Base 8)

0, 1, 2, 3, 4, 5, 6, 7

Hexadecimal (Base 16)

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Decimal (Base 10)



$$2016 = (\mathbf{2} \cdot 1000) + (\mathbf{0} \cdot 100) + (\mathbf{1} \cdot 10) + (\mathbf{6} \cdot 1)$$

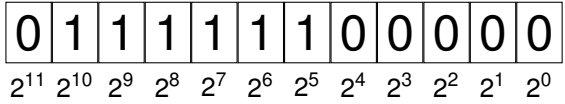
Hexadecimal (Base 16)

$$0x\boxed{7}\boxed{E}\boxed{0}$$

$16^2\ 16^1\ 16^0$

$$2016 = (\mathbf{7} \cdot 256) + (\underbrace{\mathbf{14}}_{\text{E}} \cdot 16) + (\mathbf{0} \cdot 1)$$

Binary (Base 2)



$$2016 = (\mathbf{1} \cdot 1024) + (\mathbf{1} \cdot 512) + (\mathbf{1} \cdot 256) + (\mathbf{1} \cdot 128) + (\mathbf{1} \cdot 64) + (\mathbf{1} \cdot 32) + (\mathbf{0} \cdot 16) + (\mathbf{0} \cdot 8) + (\mathbf{0} \cdot 4) + (\mathbf{0} \cdot 2) + (\mathbf{0} \cdot 1)$$

Recall

with 4 bit, you can store 16 values

```
0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
```

That's exactly how many digits we need for one hexadecimal number!

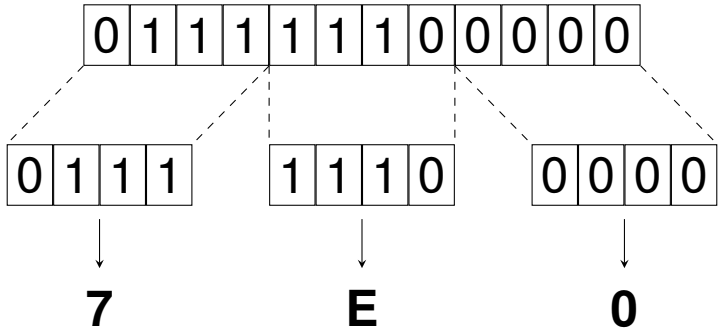
$$(\mathbf{0} \cdot 8) + (\mathbf{1} \cdot 4) + (\mathbf{0} \cdot 2) + (\mathbf{1} \cdot 1) = 5$$

Decimal ⇔ Hexadecimal ⇔ Binary

dec	hex	bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111

dec	hex	bin
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Binary ⇔ Hexadecimal



- $\Rightarrow$ `0111 1110 0000` = **`0x7E0`**