Self-Paced C++

Ad Astra Education

Dov Kruger

May 24, 2018

Many people have contributed to this work. There is the feedback of colleagues and teachers, and the help of spring and summer interns whose credits are listed below.

Authors

Dov Kruger, primary author

Editors

Bezalel Pitinsky Chanan Klapper

Layout and LaTeX Programming

Bezalel Pitinsky

Contributors

Bezalel Pitinsky

Example Programs in C++

Dov Kruger
Chanan Klapper

Online Quiz Development and Testing

Joel Hyman
Yijin Kang
Bezalel Pitinsky
Chanan Klapper

Lab Photos and Testing

Contents

C++ Programming Skills Overview

There are many things you need to know to become a good programmer. This overview lists many individual topics you need to know. You will be asked during the course to evaluate how well you know each topic. Keep reviewing this section periodically to keep track of how far you have come, how much you have learned, and which skills you have not yet mastered.

Each one of these skills is relatively limited and not a big deal. However, cumulatively you will have to study a lot to learn them all.

Category	***Skill***	***Explanation***
integer data types	signed and unsigned behavior	Whole numbers used for counting, loops.
	constants	In order to use integers, you must be able to correctly store values
	arithmetic operators	Basic arithmetic in C-like languages is similar to math, but not quite the same. Learning the limitations and differences is key to programming correctly. If you do not know what the computer does, you cannot possibly write a program that requires computing with whole numbers
	overflow	Computers have a finite amount of memory, so every number is assigned a finite amount of space. If an answer is too big it will not fit, and an overflow error will result. Understanding this is essential because it is so easy to compute values that are too big. When computation goes wrong with integers, this is something we must suspect.
floating point data types	properties	How floating point numbers behave
	roundoff	How errors can accumulate in floating point answers
	Infinity and NaN	IEE 754 Floating Point Mathematics behavior involving infinity
	math library functions	The most frequently used functions in the math library
strings	literals	How string constants are embedded in C++ programs
	string data type	The C++ library contains a string object
	string operations	What can you do to strings?
C++ Syntax	identifiers	This is the generic name for entities that have names in C++
variables	initialization	How to put values into variables when they are created
	scope	Where an identifier is visible within the program
	lifetime	How long a variable lives.
Programming Logic	decision-making (comparison operators)	Used in many statements in C++

Statements	The syntax of statements in (C/C++/Java)	What all statements share in common
	Rules of curly braces and semicolons	Defining statements and blocks
	While loop	Statement that keeps going while true
	Analyzing how many times a loop will execute	You must be able to trace code and determine what it does
	Maximum and minimum number of times a loop will execute	Key skill to avoid bugs – thinking what could happen
	If statements	Decision making statement
	do...while	Loop that executes at least once
	break and continue	Breaking out of loops early, or skipping one iteration
	goto	Uncontrolled and dangerous, but useful for breaking out of multi-level loops
	switch	A multi-way if statement that works only for integral types
	Tracing through code: mental model of what is happening	You can use a debugger, but it is quicker if you can see what your code is doing
Arrays	declaration and initialization	How to create arrays and fill them with values
	Multi-dimensional arrays	Behavior of 2, 3, 4 and higher dimensional lists
IO	printing to the screen	The cout object
	reading from the keyboard	The cin object
	writing to error stream	understanding redirecting files and why there is an error stream
	opening/closing files	understanding buffers, and the fact that if you do not close the file it may not write out, or write incompletely
Standard Algorithmic Patterns	Writing a loop to count	Understanding variables, state, tracing through a basic loop
	Summation	Knowing to initialize to zero, and understanding that a second variable is needed
	Products	Computing a product requires initializing to 1
	Array processing	Being able to perform the same kind of computation but with an array
	Nested loops	Understanding what happens when one loop is inside another
	Convergence	Computing a summation until it converges, not a set number of times
	Reading input until a value is entered	A basic pattern that shows the advantage of a do... while

Statements	The syntax of statements in (C/C++/Java)	What all statements share in common
	Rules of curly braces and semicolons	Defining statements and blocks
	While loop	Statement that keeps going while true
	Analyzing how many times a loop will execute	You must be able to trace code and determine what it does
	Maximum and minimum number of times a loop will execute	Key skill to avoid bugs, thinking what could happen
	If statements	Decision making statement
	do...while	Loop that executes at least once
	break and continue	Breaking out of loops early, or skipping one iteration
	goto	Uncontrolled and dangerous, but useful for breaking out of multilevel loops
	switch	A multiway if statement that works only for integral types
	Tracing through code: mental model of what is happening	You can use a debugger, but it is quicker if you can see what your code is doing
each object's relationships		

Intro to C++

The hardest thing in learning to program a new technology is starting. Everything is new and unknown. Nothing works at first.

A C++ program starts with a function called main. The first program we write prints out a message. To print, we send a sequence of characters, called a "string," to "cout" which represents the output.

```
      cout << "hello";
```

prints hello.

There is a lot of syntax involved in C++ and the slightest mistake will yield cryptic errors. For example, all simple statements must end in a semicolon.

In Java, everything is an object. You cannot write a program without writing a class to do it. C++ by contrast is objectcapable. You can still write C programs in C++ that do not use objects.

C++ code may only be written inside a function. The function has a name (in this case, main), parentheses after the name and then curly braces.

There is also a return type, in this case int, that will be discussed later.

Write the function main, which is the entry point in C and C++. The following program displays the classic message "Hello World". The`\n` character goes to a new line

```
int main() {
  cout << "Hello World\n";
}
```

The above program still is not quite right. The variable cout has not been defined. In order to use it, we must include a header file describing what it is.

To print in C++, include the iostream file using #include (more on this later)

```
#include <iostream>
int main() {
  std::cout << "Hello World\n";
}
```

This time, notice we are using the full name of cout, which is std::cout. This program will work.

Since it is annoying to type std:: in front of many varianles and functions, we can use a shortcut. The statement "using namespace std" tells the compiler that any time it does not understand a symbol, it should try that symbol with std:: in front of it.

```
#include <iostream>
using namespace std;
int main() {
  std::cout << "Hello World\n";
}
```

There are many named entities that we use from the std library. Typing std:: all the time can be inconvenient. So we can declare that we wish to try any unknown symbol with a std:: in front of it to see if it is in the standard library. The following program prints to the standard output object std::cout, reads from the keyboard using std::cin without mentioning std:: due to the statement: using namespace std;

```
#include <iostream>
using namespace std;
int main() {
  cout << "Please type a number:";
  int x;
  cin >> x;
  cout << " doubling: " << x * 2 << "\n";
}
```

Each compiler has its own rules. If you are using an integrated development environment (IDE) learn how to use it separately. However, there are some rules that are universal. C++ compilers expect C++ code to be in files ending with the suffix .cc, .cpp, or .cxx. Anything else will not work; the compiler will not know the program is C++.

To build C++ programs on a low level, use a compiler. From the command line:

g++ HelloWorld.cc

will generate a program called a.out on Linux and MacOSX, and a.exe on Windows. To generate an executable with the same name as the source code:

```
g++ HelloWorld.cc -o HelloWorld
```

File I/O

By default, C++ has three files open: cout which writes text to standard output, usually the terminal where the programmer is running the program, cin which reads in whatever is typed to the program, and cerr, which is the same as cout unless cout is redirected to a file. The cerr file is used to print to the screen even if the output is redirected. For example:

```
#include <iostream>

int main() {
  cout << "hello\n";
  cerr << "this is going to cerr!\n";
}
```

if this program is run:

```
g++ output.cc
./a.out
```

the output is:

```
hello
this is going to cerr!
```

On the other hand if the program redirects output to a file

```
./a.out >test.txt
```

then only hello appears. The string ”this is going to cerr!” is stored in test.txt.

C++ Data Types and constants

C++ Integer Data Types

C++supports multiple sizes of integer from small to large. The problem is, these have not been defined portably so the exact size differs depending on the computer. Also, types may be unsigned (only positive numbers) or signed (positive and negative numbers). This also affects the maximum size of the number that can be processed.

The following table shows how many bits are in each data type on a typical 64-bit computer today, and also on a small microcontroller such as Arduino (Atmel328).

Data Type	***Space (bits)***	***64-bit PC***	***Micro-controller***	***Description***
short or short int	$\geq$ 16 bits	16	16 bits	Typically signed by default.
int	$\geq$ short	32	16 bits	$\leq$ word width of machine for speed
long	$\geq$ int	64	32 bits	
long long	64 bits	64 bits	64	Guaranteed?
unsigned int	= int		32	Just like int, but unsigned so holds values twice as big
signed int				guaranteed to be signed, unlike int
char	8	8	8	Not guaranteed, but almost always 8 bits, one byte

The next table shows maximum and minimum values for each type on a 64-bit PC today. Note that there are no guarantees. The actual size is up to the compiler writer, but this is the size as reported by g++ which is what we use in this class.

Data Type	***Space (bits)***	***M***	***Micro-controller***	***Description***
short	16 bits	16	16 bits	Typically signed by default.
int	$\geq$ short	32	16 bits	$\leq$ word width of machine for speed
long	$\geq$ int	64	32 bits	
long long	64 bits	64 bits	64	Guaranteed?
unsigned int	= int		32	Just like int, but unsigned so holds values twice as big
signed int				guaranteed to be signed, unlike int

The limits for all data types are found in the header file climits. The following code illustrates:

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

int main() {
    cout << "number of bits in a byte:\t" << CHAR_BIT;
    cout << "\nsmallest number in a byte:\t" << SCHAR_MIN;
    cout << "\nlargest number in a byte:\t" << SCHAR_MAX;

    cout << "\nsmallest number in a short:\t" << SHRT_MIN;
    cout << "\nlargest number in a short:\t" << SHRT_MAX;

    cout << "\nsmallest number in an int:\t" << INT_MIN;
    cout << "\nlargest number in an int:\t" << INT_MAX;

    cout << "\nsmallest number in a long:\t" << LONG_MIN;
    cout << "\nlargest number in a long:\t" << LONG_MAX;

    cout << "\nlargest number in an unsigned long:\t" <<
      ULONG_MAX;

    cout << "\nsmallest number in a long long:\t" << LLONG_MIN;
    cout << "\nlargest number in a long long:\t" << LLONG_MAX;
}
```

The output on Ubuntu using g++ 5.4 is:

```
number of bits in a byte: 8
smallest number in a byte: -128
largest number in a byte: 127
smallest number in a short: -32768
largest number in a short: 32767
smallest number in an int: -2147483648
largest number in an int: 2147483647
smallest number in a long: -9223372036854775808
largest number in a long: 9223372036854775807
largest number in an unsigned long: 18446744073709551615
smallest number in a long long: -9223372036854775808
largest number in a long long: 9223372036854775807
```

How Signed and Unsigned Arithmetic Work

To make it manageable, consider a 3-bit computer. Each bit has two possibilities 0 and 1. So there are a total of 2^8 different combinations. How do these represent numbers? The following table shows for both signed and unsigned. If signed, the leading digit represents positive or negative so the biggest number that can be represented is half the size.

Bits	*unsigned*	*signed*
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Note that for unsigned, there is no number bigger than 7, so adding 1 to 7 wraps around to 0. This is called overflow. For signed numbers, the largest positive number is 3, so adding 1 results in -4 (the biggest negative number).

For a 16 bit signed integer, this means that: $32767 + 1 \rightarrow -32768$ (overflow). $-32768 - 2 \rightarrow 32766$ (underflow).

What does this mean for you? When writing a program, if you suddenly see a value that should be positive, but turns negative, that is an immediate sign that overflow has happened and the value is wrong. But even if you do not observe this, that does not mean that overflow has not occurred.

Be suspicious if a value should grow, and does not. And try to figure out, in advance, when you expect overflow to happen. In other words, do not way passively and wonder when your code is not working, try to compute the maximum number expected to work, and create a test plan to make sure you are correct.

C++ constants

int	2, -5	-2100000000 +2100000000
unsigned int	2U,	4200000000
long integer	2L -5L,	-2100000000 +2100000000
long long	123456789012345678LL	
short int	2h, -52h	
char n′ (newline) ′ t′ (tab) ′ ′	′a′ ′b′ ′n′ ′x0d′ ′	
string	″hello world″	

commas are not legal in numeric constants

```
cout << 2 << '\n';
cout << -5L << '\n';
cout << 5h << '\n';
cout << 2412000000000000LL << '\n';
```

Arithmetic Operators and Overflow

Integer Arithmetic

There are five basic binary arithmetic operations in C++. Each takes two values (operands) and computes a result.

+	addition
-	subtraction
*	multiplication
/	division
%	modulo (integer remainder)

Computer integer math is the same as it is in theory, except that in theoretical mathematics, there is no limit to the size of numbers. On a computer, there is a finite amount of space devoted to storing the number, and therefore, a limit to the biggest number that can be stored.

Addition and subtraction pretty much work normally as long as the maximum sizes are not exceeded. For example

4 + 5

is 9, and

5 - 9

is -4. But when a number does not fit, the answer will not be correct. The biggest signed integer that can fit into 32 bits is 2147483647. So

```
2147483647 + 1
```

does not result in a number that is one bigger since that number cannot be represented. Instead it ”wraps around” and becomes the smallest number possible: -2147483648

Similarly, if we multiply 2147483647 * 2, the result will wrap around. This is called an overflow. Sometimes the results of an overflow are obviously wrong. You know that if you keep adding, numbers do not become negative when they get big! But the result is not always obviously wrong. You must do two things: first, figure out whether the answers you intend to compute could become too big to represent, and second, be alert to the possibility that results could be wrong. DO NOT ASSUME that your computation is correct. In fact, the opposite: a good programmer must assume that the results are wrong until proven otherwise, and design unit testing to prove that the code is correct.

```
int y = 5 / 2;
int x = 5 % 2;
int a = 52 % 5;

cout << 214 % 2 << 215 % 2 << '\n'
```

Questions

1. A program starts with x= 17 and keeps doubling repeatedly. How many times would you expect the program to be able to keep going? What would this depend on?

Exercises

1. Write a program to compute the number of seconds in your age.
2. Modify the above program to be an age of 69 years. What happens to the output? Try for 70, 71, 72, 73, 74. What happens to the number?

C++11 Portable Integer Data Types

In C++11, data types have been defined that are the same across all platforms. In order to use them, compile with a C++ compiler that support C++11, and include the library cstdint

The data types beginning with u are definitely unsigned. The size of each data type is specified in bits. Thus int8_t is a signed 8-bit integer, (-128..127) while uint64_t is an unsigned 64 bit integer. Sizes 8, 16, 32, and 64 are supported.

```
#include <cstdint>
uint64_t a;
int32_t b;
```

Questions

1. Why use the portable types uint32_t and uint64_t?
2. Why was C++ (and C before it) not portable to begin with? What is the advantage of making int vary in size?

Exercises

1. Write a program that uses a portable, unsigned 64 bit integer. It should start with 1 and double until the number overflows, printing out the results one per line: 1, 2, 4, 8, 16....

Take the quiz

Type bool and Comparison Operators

The data type bool can have values false and true. It takes a single memory location (usually a byte on most computers). The value true is internally 1 and the value false is 0, but prefer using true and false for clarity.

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

int main() {
  bool b = false;
  cout << b;
  b = !b; // reverse the truth value of b
  cout << b;
  cout << (2 < 3);
  cout << (2 > 3);
  cout << (2 <= 3);
  cout << (2 >= 3);
  cout << (2 == 3);
  cout << (2 != 3);
}
```

Logical Operators

Boolean operations can be combined using logical operators. The not operator ! reverses the truth value, so for example:

```
bool b = ! (2 < 3);
```

is false because 2 is less than 3 (true), and the opposite of true is false.

The and operator can be written either with two ampersands && or as the word and. For example, in the following code snippet b1 is true because both conditions are true, while b2 is false because the second is false. Both conditions must be true for and to be true.

```
bool b1 = 2 < 3 && 3 < 4; // true
bool b2 = 2 < 3 && 5 < 2; // false
```

The logical operators in the C family of languages are short-circuiting. For an *and* operation, this means that if the first condition is false, the compiler knows the whole thing must be false, so it never tests the second condition at all. For an *or* operation, if the first condition is true, the compiler can infer the entire expression is false so the second condition is similarly never tested.

In the following example, the function f() and g() are never called because the first part of the and condition is true.

```
bool f(int x) {
  cout << "hello";
  return x % 2 == 0;
}

bool g() {
  cout << "test";
  return true;
}

int main() {
  bool b1 = 6 < 2 && f(4);
  bool b2 = 2 < 3 || g(5);
```

Questions

1. What is a bool variable? What values can it have?
2. What is true ?
3. What are the values of the following expressions:

```
2 < 3 && 3 < 4
2 < 4 && 4 <= 3
1 < 2 || 2 > 4
```

Exercises

1.

if statements

The if statement allows a statement to be executed if a condition is true (conditionally) or to execute either one or another with the else clause.

```
int x = 2;
if (x == 2)
  cout << "A"; // this is printed
```

```
int x = 3;
if (x == 2)
  cout << "B\n";
else
  cout << "C\n"; // this is printed
```

To execute more than a single line in an if statement or else clause, surround by curly braces:

```
int main() {
  int x;
  cin >> x;
  if (x >= 2) {
    x = x * 3 + 5;
    cout << x << '\n';
  } else {
    x = x / 2 + 3;
    cout << x << '\n';
  }
}
```

Variables

The result of computations can be stored in variables.

Variables must be declared.

A declaration consists of a type, a variable name and (optionally) initialization. You should almost always initialize variables.

Variables declared within functions that are uninitialized have undefined (somewhat random) values.

Variable names must begin with a letter or underscore. They may continue with a letter, number or underscore

Examples of legal variable names

```
pi
hello
ThisIsALongVariableName
Charlie48
x
amountOfVegetables
underscore_style_name
```

Variables may not contain other characters aside from those mentioned above, may not start with a digit, and may not be a reserved word in the language. A complete list of reserved words (or keywords) is here:

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

The following names are not legal

```
4pi
for
if
$xyz
ILoveC++!
```

Good Declarations

```
int x = 5;
long long y = 5000;
char c = 97; // int will automatically convert to anything
   smaller
short int s = 256; // int --> short is not a problem
```

Bad declarations

```
int z = 500000000000000LL; // losing information (warning)
```

Questions

1. What does a strongly-typed language require for variable declarations?
2. What is the value of a variable in a function in C++ if it is not initialized?

Exercises

1. Write a program to compute the number of hours in a week. Use a variable to store the value before printing.

Divide By Zero

For integers, a divide by zero results in a hardware exception. The processor actually sends a special message to the operating system, terminating the program. This is pretty bad, since the program stops dead. If the program is a word processor, for example, then it dies without saving. If the program is running a pacemaker or other lifesaving equipment, the results could be catastrophic. For this reason, it is possible to write code to trap the failure and recover, but the best defense is whenever a division is computed that could result in a divide by zero, test the denominator first, and do not do the division if the denominator is zero.

Questions

1. What is 15125125 % 2
2. What is 151251249885624 % 2
3. What is 3/0? (Try it on your computer!)
4. What is wrong with writing a program that divides by zero?

Exercises

1. Write a program to read in the temperature in Fahrenheit and convert it to Celsius
2. Write a program to read in a number. Print out whether the number is odd or even.

Take the quiz

I/O in more Detail

The iostream header file supports three files: cout (default text output to the screen), cin (input from the keyboard), and cerr, which will still appear on the screen even if the default output file cout is redirected somewhere else.

If you are not used to using the command line, the distinction between cout and cerr may not be obvious. Here is an example showing the code, and then showing how to run the program from a unix command line. The same can be done in Windows from a windows command line.

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

int main() {
  cout << "testing testing 123\n";
  cerr << "this will show even if redirected\n";
  int age;
  cout << "Please enter your age: ";
  cin >> age;
  cout << "Your age is " << age << '\n';
}
```

The following lines show compiling the program in Linux, then running the program:

```
$ g++ io.cc
$ ./a
testing testing 123
this will show even if redirected
Please enter your age: 50
Your age is 50
```

When run with output redirected to the file log, all the regular output disappears, but cerr output stays on the screen. This is so error messages can still appear:

```
$ ./a >log
this will show even if redirected
```

```
50
```

C++ supports reading any variable types from keyboard. Reading a single char will read a single one-byte letter. Values are assumed to be separated by spaces, so spaces between numbers will be skipped. For example:

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

int main() {
  cout << "reading in int, float, double, long:";
  int a;
  float b;
  double c;
  long d;
  cin >> a >> b >> c >> d;

  cout << a << '\n' << b << '\n' << c << '\n' << d << '\n';

  char x;
  cin >> x; cout << x << '\n';
  cin >> x; cout << x << '\n';
  cin >> x; cout << x << '\n';
  cin >> x; cout << x << '\n';
}

$ ./a
reading in int, float, double, long:5 4.2 9.8765432123
   10241214159
5
4.2
9.87654
10241214159
abcde
a
b
c
d
```

Questions

1. How do you read a value from the keyboard?

2.

Exercises

1. Write a program to read in the temperature in Fahrenheit and convert it to Celsius
2. Write a program to read in a number. Print out whether the number is odd or even.

Take the quiz

Floating Point Types

We have seen the whole number (integral) data types that C++ and other languages support. But computers also have to compute decimals. Decimal types are called floating point because the decimal place can move in this representation. Floating point has a sign (+/-), a fixed number of digits of precision (callled the mantissa), and an exponent that defines how big or small the number is.

```cpp
float f = 1.5f;
double d = 2.5;
long double d2 = 1.234567890123456789012345 67;
const double avogadro = 6.023e+23;
const double pi = 3.14159265358979;
const double massEarth = 5.96e+24; // in kg
```

Questions It is a warning in C++ to lose accuracy in a computation. Can you spot where this is happening in each example?

```cpp
float f = 1.5 + 2.5;
long double PI = 3.14159265358979L;
double pi = PI;
```

C++ Data Type Summary

In C++:

Data Type	*Space (bits)*	*PC*	*Micro-controller*	*Description*
short	$\geq$ 16 bits	16	16 bits	Typically signed by default.
int	$\geq$ short	32	16 bits	$\leq$ word width of machine for speed
long	$\geq$ int	32	32 bits	
long long	64 bits	64 bits	64	Guaranteed?
unsigned int	= int		32	Just like int, but unsigned so holds values twice as big
signed int				guaranteed to be signed, unlike int
float	32 bits	IEEE754		about 7 digits precision
double	64 bits	IEEE754		about 15 digits precision
long double	opt l			optional and size depends on platform

In practice, on normal pc: short=16, int=32, long=32
on small, embedded processors (Arduino) short=16, int=16, long=32

long long = 64 bits (8 bytes) char is one byte typically, but always 1 minimal addressing unit.

float single precision

double double precision (except on Arduino, where double=float)

long double (more precision, not specified)

char and string

The C++ language has built-in support for characters. A sequence of letters is called a string and uses a class, an extension of the language. The STL (Standard Template Library) supports strings, or sequences of characters. To use it, the programmer must include the string library. String constants are stored in double quotes. Characters are a single letter in single quotes.

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

int main() {
  char c = 'a'; // char variables can hold a single letter
  string s = "hello there";
  cout << c << s << '\n'; // the newline character \n  goes to a
     new line
}
```

Many of the same operations that can be performed on integers work on strings. They can be copied using =, compared any of the comparison operators. Strings can be added using operators + and +=, but this means that the second string is concatenated (tacked onto the end).

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

int main() {
  string s1 = "testing";
  string s2 = "abc";
  cout << (s1 + s2) << '\n';
  s2 += s2;
  cout << s2 << '\n';
}
```

String Manipulation

The string class supports many operations. The following list are the most commonly used.

s[i]	the ith character in the string
s.size()	the number of characters in the string (the length)
s.substr(pos, len)	return a substring starting as pos going len characters
s.insert(pos, s2)	insert s2 into s at position pos
s.erase(pos, len)	remove the characters from pos going len characters
s.replace(pos, len, s2)	replace from pos going len characters with string s2

The following example shows the functions above in action.

```
int main() {
  string s1 = "abc";
  cout << s1.size();
  s1.insert(1, "bigger"); // s1 is now "abiggerbc"
  s1[3] = 'N'; // s1 is now abiNgerbc
  s1.replace(2, 5, "acus"); // s1 is no abacusrb
  s1.erase(6,2); // s1 is now abacus
}
```

Rules of Semicolon and Curly Braces

Every simple statement ends in a semicolon. This is because statements can span multiple lines.

Example:

```
int main() {
  x = 365.2425 * 24 * 60 * 60;
  double x = 1 + 3 / (1*2*3) + 3*3 / (1*2*3*4) +
     3*3*3/(1*2*3*4*5) +
    3*3*3*3/(1*2*3*4*5*6);s
}
```

Curly braces are used to combine multiple statements into a single compound statement. For example, the following compound statement combines three assignments:

```
{
  temp = a;
  a = b;
  b = temp;
}
```

Anywhere a single statement can go, a compound statement may be placed. Therefore, since an if statement executes a singlel statement conditionally, it can be used to execute a number of statements conditionally using curly braces.

Questions

1. What is the purpose of a semicolon in C++?

2. What do curly braces do?

Exercises

1. Write a program that prints hello, forever.
2. Write a program that counts from 1 to 10 printing each number on a separate line.
3. Write a program that counts up from 1 to 10 and down from 10 to 1 at the same time, printing 1 10, 2 9, 3, 8, ... until 10 1.

Take the quiz

Floating Point Calculations and Roundoff Error

C++ supports decimal calculations. These are called floating point. There are three suported types: float, double, long double.

Single precision has 7 digits precision and can range from +/- $10^{3}8$ Double precision has 15 digits precision and can range from +/- $10^{3}28$ It is stored in 64 bits (8 bytes). long double is bigger, and depends on the architecture of the machine.

The IEEE 754 standard defines the behavior of single and double precision.

You can read more detail about IEEE floating point on Wikipedia: IEEE Floating Point

```
float f = 1.0f;
double d = 1.5;
long double big = 1.50000000000000000000000000000000123L;

float g = 12345.67;
float h = 1.234567e+20;

const double AVOGADRO = 6.022e+23; // it is conventional to use
   all CAPS for constants
const double PI = 3.14159265358979;
const double G = 6.6711e-11;
const double earthMass = 5.96e+24; // mass of the earth in kg
```

General Principles

In general, always use double. Single precision float has too few digits, and the answer can end up completely wrong.

Printing out Floating Point Values

By default, floating point values will print as whole numbers if the fraction is zero, or with 5 digits precision if there are digits after the decimal. For more control, include the iomanip header file. This provides two objects to control output: setprecision, which controls the number of digits displayed, and setw which controls the width of the field displayed. Whenever more digits are specified than width to display them, the digits will overflow the field rather than display the wrong number. For example:

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

int main() {
    cout << 5 << '\n';
    cout << 1.234567f << '\n';

    cout << setprecision(12) << 1.234567f << '\n';
    cout << setprecision(15) << 1.234567 << '\n';

    cout << setw(30) << 1.23 << setw(30) << 2.34 << '\n';
    cout << 1.23 << 2.34 << '\n';
}
```

Roundoff error

Floating point is inherently inaccurate. It can represent huge ranges of values, but accuracy is only relative. For example, in a 32-bit float there is 1 bit for sign, 10 bits for the position of the binary point, leaving 22 for the digits in the number (called the mantissa). Due to a hardware trick, we get an extra bit, so it is effectively 23 bits for the mantissa.

Floating point numbers are expressed in base 2 because they are stored on a digital computer where each bit is 1 or 0. This means that the mantissa contains powers:

2^{-1},2^{-2},2^{-3},2^{-4}...

The first digit of the number is $\frac{1}{2}$, the second is $\frac{1}{4}$, the third is $\frac{1}{8}$, and so on. So for a floating point number with the following mantissa:

101000000000000000

the number is $\frac{1}{2} + \frac{1}{8} = 0.625$.

There are a number of problems. You know from basic arithmetic that there is no way for you to express the fraction $\frac{1}{3}$. That is because we use base 10, and 3 does not divide evenly into 10. The same is true in base 2, $\frac{1}{3}$ is a repeating fraction. However, you are used to the fact that $\frac{1}{10}$ is a "nice" fraction, and in base 10 we write it as: 0.1.

In binary, however, $\frac{1}{10}$ is a repeating fraction: $\frac{1}{16} + \frac{1}{64} + \frac{1}{256} + ...$ so internally the fraction is represented as: 0.0010101010101010101...

The following program illustrates the problem. Print out numbers from 0 to 10 stepping 0.1 and very soon, they start to be wrong, and they get worse the longer you go.

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

int main() {
    for (float x = 0; x < 10; x += 0.1)
        cout << x << ' ';
}
```

```
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2
2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4
4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 6
6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9 7 7.1 7.2 7.3 7.4 7.5 7.6 7.7 7.79999 7.89999 7.9999
8.09999 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9 9 9.1 9.2 9.3 9.4 9.5 9.6 9.7 9.8 9.9
```

Questions

1. What is floating point?
2. When should float be used (as opposed to double)?

Exercises

1. Write a program that reads in scores from the keyboard. Numbers should be between 0 and 100. End by typing in a negative number and print out the average.
2. Write a program that reads in two sides of a triangle and computes the third side using the pythagorean theorem $c^2 = a^2 + b^2$.
3. Write a program that reads in three sides of a triangle and computes the area using Heron's formula:$s = \frac{a+b+c}{2}, A = \sqrt{s(s-a)(s-b)(s-c)}$.

Roundoff Error

Some numbers are "nice" in decimal. Fractions like 1/2 = 0.5 or 1/10 = 0.1 can be exactly represented. Computers represent fractions in binary. Floating point fractions are sums of 1/2, 1/4, 1/8, ... Some fractions we think of as "nice" like 1/10 cannot be exactly represented in binary.

1/10 in decimal is 0.000101010101..... in other words, 1/10 = 1/16 + 1/64 + 1/256 + ...

You can see this by counting from 0 to 10 by 0.1. Very quickly, the numbers get a little off. They get more and more wrong the more is added. This is called roundoff error.

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

int main() {
  for (double x = 0; x < 10; x += 0.1)
    cout << x << '\n';
}
```

When printing the numbers from 0 to 1 stepping by 0.1, we can do this two ways:

```
#include <iostream>
using namespace std;
for (double x = 0; x <= 1.0; x += 0.1)
  cout << x << ' ';
cout << '\n';
for (int i = 0; i <= 10; i++, x += 0.1)
  cout << x << ' ';
}
```

The output of the above program is:

```
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2
```

Questions

1. Why are the operators == and != almost never used for floating point?
2. What is the value of 0.1 + 0.2 + 0.3 + 0.4 + 0.5 == 1.5
3. Why are the two loops different in the program above?

Exercises

1. Write a program to read in a radius and print the area of a circle of that radius.
2. Write a program to read in an x,y location which is the center of a circle and the radius. Then read in x,y points and print either "inside" or "outside" depending on whether the point lies in the circle or not.
3. Write a program to sum 1/1 + 1/2 + ... + 1/100. Then do the sum in reverse and compare the answers (print the difference). They are not the same, why?

For a great example of how roundoff affects real life, see this article on the Patriot missile software problem.

The C++ Math Library

The C++ math library is in cstdlib. Older C++ and C used math.h, but this is now obsolete. The following table shows the most common functions in the math library

All functions are computed in double precision unless otherwise specified

sin(x)	sine
cos(x)	cosine
tan(x)	tangent
atan(x)	arctangent
atan2(y,x)	arctangent(y/x)
sqrt(x)	square root
exp(x)	e^x
log(x)	$log_e(x)$
pow(x,y)	x^y

Incredibly, the math library does not have a standard definition for pi, which boggles the mind after all these years. Instead, define:

const double PI = 3.14159265358979;

and use that for most purposes.

The functions above are the most commonly-used in cmath. All trigonometric functions are in radians, so for angles in degrees, convert the angles to radians.

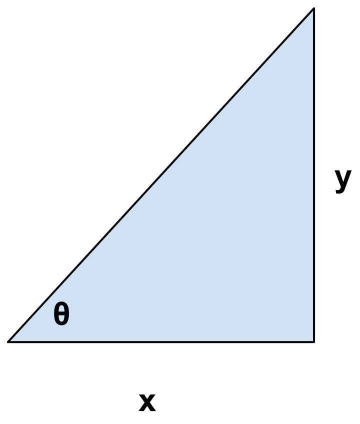
```
const double DEG2RAD = PI/180; // see above for PI!
double y = sin(x * DEG2RAD);
```

to convert from a function returning an angle, do the opposite:

```
const double RAD2DEG = 180/PI; // see above for PI!
double y = atan(x) * RAD2DEG;
```

Special Cases

One very common trigonometry problem is computing the angle when x and y are known. It comes up in robotics when the computer knows the destination (or target) coordinate, and its own coordinate, and needs to compute at what angle to head.



In the case above, the equation of the angle is $\theta = tan^{-1}(\frac{y}{x})$. However, never use this on a computer, because there are many special cases starting with the one where x = 0. Instead, C++ (and most other languages) have a library function called atan2 which takes y and x as separate parameters and correctly handles all cases:

```
double y = atan2(y, x);
```

As usual, if the answer is to be in degrees, convert it back from radians.

Questions

1. Why should you never compute atan(y/x)?
2. How do you compute sin(30 degrees)?

Exercises

1. Write a program that reads in a distance and an angle θ in degrees. Given a starting coordinate of (0,0) determine where travelling for that distance at that angle would end up. The equation is $x = rcos(\theta), y = rsin(\theta)$.
2. Modify the previous program to accept a list of distances and angles and display the position at each step until the end.

IEEE Floating Point: Infinity and NaN

Floating point values can perform operations like divide by zero. Rather than crashing the program (as integer operations do by default) the IEEE standard defines results that makes sense numerically and in calculus. As the denominator approaches zero, the result of a division tends toward infinity. Therefore, the IEEE standard defines a special value, infinity to represent the result of this calculations.

Infinity is not like other values. Infinity + 1 is still infinity. Infinity * 2 is still infinity. And when infinite values battle it out, we cannot tell what the result will be, so Infinity - Infinity is not zero. In such cases, the IEEE standard defines another value: NaN (Not a Number). The NaN value is used when the value of a computation is indeterminate.

The following cases all generate positive and negative infinity:

```
#include <iostream>
using namespace std;
int main() {
  double x = 0.0;
  double y = 1 / x;
  double z = -1 / x;
  cout << y << z << '\n';
  cout << (y+1) << '\n';
  cout << (z*1) << '\n';
}
```

Dividing a finite value by infinity yields zero. Since there is positive and negative infinity, this leads to the concept of positive and negative zero. Even though the sign shows that they come from opposite infinities, if they are compared, they are equal to each other.

```
  double y = 1 / x;
  double z = -1 / x;
  cout << 5 / y << '\n';
  cout << 5 / z << '\n';
  cout << "positive zero and negative zero: " << (5/y == 5/z) <<
     '\n';
```

Last, when two opposing infinite forces meet, we don't know what the result is. Whether this is because of a zero in the numerator battling with a zero in the denominator, or +infinity fighting with -infinity, all the results below are NaN.

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
  double x = 0.0;
  double y = 1 / x;
  double z = -1 / x;
  cout << (y+z) << '\n';
  cout << x / x << '\n';
  cout << sin(y) << '\n';
  cout << sqrt(-1) << '\n';
}
```

Functions

Functions are named chunks of program. They are used to break up programs into manageable sections. Every c++ program we have written has a function called main(), which is the standard entry point (where C++ starts). But you can write functions to break up a big complicated program.

The first example function will just print out something.

```
void f() {
  cout << "hello";
}
```

Notice that a function has a name (in this case, f) but it has parentheses after the name to indicate that it is a function not a variable. So with two declarations:

```
int x;
int y();
```

x is an integer variable, while y is a function (that returns an int value). In order to call the function from main we type the name followed by parentheses. The function must be known before the compiler sees the call, or there wil be an error. In the example bellow, the function is define above main so the compiler knows what the name f is:

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

void f() {
  cout << "hello";
}

int main() {
  f();
}
```

The above function is not that useful because it always prints the same thing every time. In order to make functions more useful we add parameters, values that can be sent to the function. The parameters are specified within the parentheses. Each parameter must have a data type, and a name. The following example shows a function f that accepts an integer parameter, and a function square that accepts a double parameter.

```
void f(int x) {
  cout << "The value passed in is: " << x << '\n';
}
void square(double v) {
  cout << v * v;
}

int main() {
  f(2);
  f(3);
  square(4);
}
```

Questions

1. What is a function?
2.

Exercises

1. Write a function that has two double parameters representing the sides of a triangle and computes the hypotenuse. Your program should read in two values and compute the third.
2. Write a function that takes four double parameters:

   ```
   bool contains(double xcenter, double ycenter, double radius,
       double x, double y)
   ```

   The function should return true if the circle centered at xcenter,ycenter with radius contains point (x,y).

if Statement

An if statement checks whether a condition is true or false. If the condition is true, it executes the statement immediately after the if statement. If the statement is false, it either executes nothing, or if there is an else clause with a second statement, it executes that one.

Example:

```
if (c)
  statement1;

if (c2)
  statement1;
else
  statement2;
```

In the following example, the program prints ”yes” and ”no” because the first if statement is true and the second is false.

while Loop

Structure

The while loop looks almost like an if statement. It too has the parentheses surrounding the condition. If the condition is true, it execues the statement immediately after the while statement. However, in addition it then checks if the condition is true again. While loops will go forever if nothing changes the condition. They can also execute zero times if the condition is initially false.

```
  while (2 < 3)
    cout << "Yes"; // this always prints  "yes" in an infinite
       loop (2 is always < 3)
```

The following while loop never executes because the condition is initially false:

```
  while (3 < 2)
    cout << "Yes"; // nothing is printed
```

Counting Loops

In order to count, use a variable and add until it reaches the desired number

```
int count = 0;
while (count < 10) {
  cout << count;
  count++;
}
```

The above loop first checks whether 0 ¡ 10 (it is). Then it prints 0 and count becomes 1. The last value printed is 9.

The next loop is similar, but x starts with 1 and counts to 10.

```
int x = 1;
while (x <= 10) {
  cout << x;
  x++;
}
```

There are four ways of adding 1 to a variable in C++. Get to know all four because they are commonly used.

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

Questions

1. Why are loops used in programming?
2. What is the least number of times a while loop can run?
3. What is the highest number of time a while loop can run?

Exercises

1. Identify the output of each of the following loops:

```
int count = 5;
while (count <= 10) {
  cout << count;
  x = x + 2;
}
```

```
int count = 10;
while (count > 0) {
  cout << count;
  count--;
}
```

2.

for Loop

The for loop is shorthand for a while loop with built-in place to put the code to initialize a variable, and to change it to the next value. The following while loop is exactly equivalent to the for loop. The for loop is more compact and has other advantages, so is generally better to use for typical counting loops.

The for statement has three expressions within the parentheses:

```
for (init-expr; while-condition; next-expr)
   statement
```

The init-expr is executed once. The while-condition is tested each time before the statement (the body of the loop) is executed. The next-expr is executed each time at the end of the loop.

A for loop is more compact than a while loop, but it is exactly equivalent. Here are two loops, a for and a while, both of which count from 0 to 9.

```
for (int x = 0; x < 10; x = x + 1)
   cout << x;
```

```
int x = 0;
while (x < 10) {
   cout << x;
   x = x + 1;
}
```

Loops do not have to count up linearly. The expressions just change the values of the variables until they end. See if you can following the logic of the following loops.

```
for (int x = 10; x > 0; x--)
   cout << x;
```

```
for (int x = 1; x  < 100; x *= 2)
   cout << x;
```

```
for (int x = 1; x  < 10; x += 2)
   cout << x;
```

```
for (int x = 1; x  < 10; x -= 2) // not an infinite loop, but
   very, very long!!! 1, -1, -3, -5, -7, ... -MAXINT, overflow
   , positive (stops)
   cout << x;
```

do ... while Loop

A while loop checks the condition at the beginning, so it is possible to execute the loop zero times (the condition fails immediate). For example, the following while loop does nothing:

```
   while (false)
      cout << "print nothing";
```

The do... while loop checks the condition at the end rather than the beginning like the while loop. It is not used as much as the while loop, but it is useful whenever you need to do something first, then go back if it is not done correctly. For example, read in a number from the keyboard, and go back and read again if the number is not acceptable. The following example keeps reading in values until one is greater than or equal to 10.

```
int main() {
  int x;
  do {
    cin >> x;
  } while (x < 10);
 // by the time control reaches this point, x must be bigger than
    10
}
```

break Statement

The break statement is used to break out of a statement. In this section we will just look at using it to break out of loops from the middle. Note that it is easy to construct weird logic that is difficult to understand. Ideally, a loop has a single exit condition (the while clause) but it is not always possible.

```
int n = 17;
for (int i = 2; i  < n; i++)
  if (n % i == 0) {
    cout << n << " is divisible by " << i < "\n";
    break; // get out of the loop if we find a divisor
 }
```

Note that if the goal is merely to get out of the loop, this could be written into the for as follows:

```
int n = 17;
for (int i = 2; i  < n && n % i != 0; i++)
  ;
```

continue statement

The continue statement skips the current iteration of a loop and keeps going with the next one. A loop containing an if statement with an if statement which executes continue is the same as executing the remainder of the loop inside an if statement with the inverse condition, as shown below:

```
for (int i = 0; i < 10; i++) {
  if (i == 5)
    continue;
  cout << i << ' ';
}
```

is exactly the same as:

```
for (int i = 0; i < 10; i++) {
  if (i != 5)
    cout << i << ' ';
}
```

goto statement

At the lowest level, computers implement loops by determining whether a condition is true and jumping to a different place in the program. The while and for loops are higher level ways of looking at loops that are clearer. C++ does provide a goto statement mirroring the low-level assembler code that computers really run. While goto can do things that structured loops cannot do, it is extremely confusing and easy to write completely unreadable code. The following example shows a simple loop created using an if statement and a goto. This may not look too bad, but never, ever use this.

```
  int x = 0;
loop:
  cout << x << ' ';
  if (x < 10)
    goto loop;
```

To illustrate how confusing gotos can be, consider the following spaghetti code, so called because there are gotos in multiple directions, and it is hard to see where one strand begins and the other ends.

```
  int x = 0;
start:
  cout << x << ' ';
  x--;
middle:
  if (x % 2 == 0)
    goto cond;
  cout << x+1 << ' ';
  if (x % 3 == 0) {
    x += 4;
    goto start;
  }
cond:
  if (x < 10)
    goto loop;
```

Essentially, there is only one case where goto should be used in C++. That is when you have a nested loop and want to break out of the middle. There is nothing else that does that as cleanly as goto. The following example shows a prime number loop which is repeatedly testing a number n mod i. If a divisor that evenly multiplies n is found, then the number is definitely not prime. However, if the entire inner loop executes and none of the numbers are divisors of n, then n is prime. The goto in this case skips the prime check in a simple, clean way. Any other method is more complicated.

```cpp
int count = 0; // so far, we have found no prime numbers
for (int n = 2; n <= 1000; n++) { // find all numbers from 2 to 100 which are prime...
  for (int i = 2; i < n; i++)
    if (n % i == 0)
      goto notPrime;
  count++; // if the entire inner loop passes, then the number must be prime
notPrime:
  ; // this empty statement is where the test jumps to if the number is NOT prime
}
```

Note that in languages like java where the continue statement can be labelled to apply to the outer loop, this particular example is not needed either. Here is a way of writing this in C++ without the goto. Note that we require a flag to remember whether we successfully divided the number or not. It's slightly slower, which is not a big deal, but also less clear.

```cpp
int count = 0; // so far, we have found no prime numbers
for (int n = 2; n <= 1000; n++) { // find all numbers from 2 to 100 which are prime...
  bool prime = true; // let's assume for the moment that n is prime
  for (int i = 2; i < n; i++)
    if (n % i == 0) {
      prime = false;
      break;
    }
  if (prime) // at the end of the loop, check whether the flag is true or not.
    count++;
}
```

switch statement

The switch statement is a weird multi-way if statement with some odd properties. First, it is equivalent to testing for equality with each case.

```
int x;
cin >> x;
switch(x) {
case 1:
  cout << "one ";
  break;
case 2:
  cout << "two ";
  break;
case 5:
  cout << "five ";
  break;
default:
  cout << "something else";
  break;
}
```

The above code is equivalent to an if...else chain as follows:

```
int x;
cin >> x;
if (x == 1) {
  cout << "one ";
} else if (x == 2) {
 cout << "two ";
} else if (x == 5) {
  cout << "five ";
} else {
  cout << "something else";
}
```

The break statements are necessary in the switch statement. If not, control would continue downwards. So without the break:

```
int x;
cin >> x;
switch(x) {
case 1:  cout << "one ";
case 2:  cout << "two ";
case 5:  cout << "five ";
default:  cout << "something else";
}
```

```
float x = 1;
switch (x) { // ILLEGAL!!! switch value must be integral type
case 1: cout << x;
}
```

the above code, which tests whether $x == 1$ would first print "one ", and then print "two " and then "five ", and finally, "something else ". In other words, it does not get out after each case, it keeps on going from that point. Each case is actually a goto label.

The other odd quirk of the switch statement is that it tests for equality, so it is illegal to use a floating point value because (as we have learned with roundoff error) floating point values are rarely equal to anything. For example, the following code adds 0.1 to sum 100 times. This should be 10.0. But because of roundoff error, it is not quote 10.0, and therefore a test for equality would fail. Knowing that this is almost never a good idea, the switch statement only supports integer types (short/long, uint64_t, char, etc, but not float, double, or long double).

Recursive Functions

A function that calls itself is recursive. All recursive functions must contain an if statement to terminate the recursion. Otherwise, they will keep calling themselves, taking more memory until the computer runs out.

The simplest kind of recursion is called tail recursion because the call happens at the end. Here is a recursive definition of factorial which is based on the rule that 0! = 1 and for all n, n! = n * (n-1)!

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  return n * factorial(n-1);
}
```

More complicate recursions can involve the function calling itself multiple times. In this case, because fibonacci calls itself twice, as the parameter n grows, the function is called 2^n times.

```
int fibo(int n) {
  if (n <= 2)
     return 1;
  return fibo(n-1) + fibo(n-2);
}
```

Arrays

Arrays are variables that hold lists of numbers indexed by position. Every element in a C++ array must be the same type. The size is declared with square brackets. For example, the folllowing declares an array a of 10 integers, and an array b of 5 doubles.

```
int a[10];
double b[20];
```

Just like variables with a single value, arrays declared in a function are not initiaized (they are random junk). To initialize an array, use the equal sign, but enclose the values within curly braces. If even one value is specified, all the remaining values in the array will automatically be initialized with 0.

```
int a[10] = {1,2};
double b[20] = {1.5, 3.5, 7.2 };
```

The first element of a C++ array is located at position 0. The following program prints out 1.5, then 1 then 0.

```
int main() {
  int a[10] = {1,2};
  double b[20] = {1.5, 3.5, 7.2 };
  cout << b[0] << '\n';
  cout << a[0] << '\n';
  cout << a[3] << '\n';
```

In order to process an array in a loop, use a for loop that goes from 0 to one less than the size. Remember that since the array starts with element 0, the last element is *not* n but n-1. It is also useful to declare a constant so that the size of the array and the loop are guaranteed to be the same, as shown in the followig example:

```
#include <iostream>
using namespace std;
int main() {
  const int SIZE = 5;
  int a[SIZE] = {9, 8, 5, 2, 6};
  for (int i = 0; i < SIZE; i++)
    cout << a[i] << ' ';
```

Questions

1. What is an array?
2. What is the first element position in an array?
3. What is the last element position in an array with n elements?

Exercises

1. Write a program to read in a number. Then read in that many x,y,radius. Once you have read in all the circles, read in a new point x,y and determine whether it is in any of the circles, and print "inside" or "outside." Repeat untill the user stops typing in input.
2. Write a program to read in grades from a file. The grades should be between 0 and 100. Read all the numbers into an array and then compute the average and variance. Make sure your array is big enough to hold the largest conceivable file of data because we have not yet covered how to handle a list that has to grow.
3. Is an array necessary to compute the average? *** Need more problems to compute!

inline Functions and Performance

In C++, unlike Java, almost everything must be declared before used. double f(double x); // there is a function, somewhere, called f

```
int main() {
  cout << f(2); // this will work provided f(x) actually exists.
}
```

The C calling convention: In order to call a function:

1. push the parameters
2. jsr (jump to subroutine)
3. in the function, pull the values off the stack. do computation
4. rts (return from subroutine)
5. caller copies the return value somewhere
6. add to stack pointer (pop the parameters off the stack)

Calling lots of subroutines is slow because the processor must push parameters on the stack, go somewhere, then pull them off the stack. Only then is the computation performed.

Object-oriented code typically contains many function calls so reducing this overhead can dramatically improve performance. inline code optimizes some of this out of existence inline is just a suggestion, but the compiler will try.

When a function is declared with the keyword inline the compiler will attempt to inject the body of the function at the call site instead of callling anything. In other words, instead of going somewhere, a copy of the code is pasted every time the function is called. For small functions this can be dramatically faster. The following is an extreme case. The code below defines a function f that takes a single parameter and returns the square. For the case of passing 2, this would result in about 10 machine language instructions. However, because it is inline, the compiler replaces the call to f(2) by 2*2. Since this is a constant expression, the compiler replaces it by 4. In other words, with an inline function there is no computation at all.

```
inline double f(double x) { return x*x; }

int main() {
 cout << f(2); // HUGE win, this turns into 4 (no computation!)
}
```

Good general rules for painless performance optimization

1. *always* use inline for one-liners, function that just calls another function with minimal computation.
2. *don't* request inline if the code >3 lines, if there is a huge loop, anything where the percentage of time to get in and out is tiny, and the code is bigger if generated inline.
3. *if* your code does not call the function more than once, inlining cant hurt.
4. functions cannot be inlined if the compiler does not have the code. For a function prototype, which just has the function parameters and a semicolon, inline is meaningless.

```
inline double f(double x); // cannot inline, do not know what to
   do
```

Questions

1.
2.

Exercises

1. Write code to compute the sum of the squares of the numbers from 1 to 1 biillion by calling a square function. Try this both with and without the inline and see what happens. In Unix, assuming you build the program called sum1billion you can time the result with the command:

   ```
   time sum1billion
   ```

Pass by Reference

Parameters by default are passed by value. That means no matter what you do to a parameter, it does not change the original.

New notation to pass by reference. Looks like pointers, but its different. Reuses the & symbol in a different context.

Problem: Rect to polar conversion requires r and theta back. You can only return one value Instead, use references:

```
void rect2polar(double x, double y, double& r, double &theta) {
  r = sqrt(x*x+y*y);
  theta = atan2(y,x);
}

int main() {
  double x = 3, y = 4;
  double r, theta;
  rect2polar(x, y, r, theta);
  cout << r <<    ,    << theta <<    \n   ;
}
```

1. Write the following function to compute the maximum, minimum, mean and variance

   ```
   void stats(double[] x, int n, double& max, double& min,
      double& mean, double& var);
   ```

2. Write a second version of the stats code that computes the result in a single loop (see equation in wikipedia, how to calculate variance)
3. **Wikipedia also mentions a more numerically stable way that is a bit more complicated

Pointers in C and C++

Everything in memory has an address. Pointers are the C way of declaring a variable that holds an address.

Pointers in C and C++ are typed. For example, a pointer to int can only point to values of type int.

```
int x;
int* p = &x;

int y;
int * q = &y;
```

```
float a;
int* r = &a; // error, r is a pointer to int, not float
```

To describe pointers, read them backward. For example:

```
int a;
int* p; // p is a pointer to int
p = &a; // make p point to a
```

There are two kinds of const-ness for pointers. A pointer may be readonly, meaning it can look at a position in memory but not write there. The clearest way is to read the declaration backwards. For example:

```
int a;
const int* b = &a; // b is a pointer to int that is const (a
   readonly int)
```

On the other hand, if the const comes after the pointer, it means that the pointer itself is constant and cannot point to any other place in memory. By default, pointers may be assigned anywhere:

```
int a = 1, b = 2;
const int* p = &a; // p now points to a
p = &b; // now p points to b
cout << *p << "\n"; // prints out 2
```

In order to lock the pointer p to a location, declare it to be constant as follows:

```
int a = 1, b = 2;
int* const p = &a; // p is a const pointer (one that cannot move)
    to int
p = &b; // error, p is a constant pointer to int, cannot change
```

To look at what a pointer is pointing at is called *dereferencing* the pointer. This is done using the operator *. For example:

```
int a = 1;
int * p = &a; // p points to a
cout << a << "\n";  // print out the value of a
cout << *p << "\n";  // print out the value that p is pointing to
    (same as a)
```

Pointer Arithmetic

The layout of variables in memory is implementation defined. Do not assume that just because two variables are declared next to each other, that they will be adjacent in memory.

However, arrays are guaranteed to be a contiguous block of memory

Arrays are equivalent to a constant pointer to the beginning of a block of storage.

```
int a[] = {1, 3, 5, 7, 9};
int* const b = a; // b and a are effectively the same.  Neither
   can move
```

We can do arithmetic on pointers. In the above example, a and b both point to the first element of the array. The expression:

```
a+1
```

points to the second element of the array (to the number 3). To access that element:

```
*(a + 1)
```

In general, for any i, the above is equivalent to array lookup using the square brackets. So the following two lines are identical:

```
k = *(a + i);
k = a[i];
```

The C++ Standard Library (Formerly known as STL)

The C++ Standard Library is a collection of classes and functions that provide many resources that are important for writing programs in C++. There are many classes such as lists (vector and linked_list) trees (map), hashmaps (unordered_map) and others. There are also algorithms which are written in a portable way so that they can be applied to many data structures. For example, the sort algorithms in STL can be used on many kinds of lists.

vector

The vector class is a list that can grow and shrink. It requires including ¡vector¿ The following example shows how to build a list, adding to the end using the push_back method. It contains both methods that view and modify the list without checking, and also methods that check whether an element is out of bounds before getting it.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> a; // a is a list of integers
  for (int i = 0; i < 10; i++)
    a.push_back(i);
  // a now has size 10.  The first element is 0, the last is 9.
  for (int i = 0; i < a.size(); i++)
    cout << a[i] << ' ';                  // access the elements
       like an array
  return 0;
}
```

The class vector is a generic class, so in addition to typing vector, you must specify the parameter that defines "vector of what?" Example:

```
vector<int> a;  // a is a list of int
vector<double> b;  // b is a list of double
vector <Elephant> elephants; // elephants is a list of elephant
   objects.
```

The class vector is very fast for appending to the end, but due to its design it is very slow for adding values to the front of the list (or the middle).

Iterators

An iterator is a design pattern that keeps track of a position within something else. Iterators are the most-used design pattern. There are iterators to walk through lists (the topic here), iterators to keep track of the position within a database (called a cursor), the position on the screen (also called a cursor). The last example showed how to track position within a vector using an integer index, but in general, an integer position is not always efficient. For another list (called a linked list) using an int to keep track of a position would be very slow. General rule: always use an iterator to keep track of position, because it will be written to do so in the most efficient way possible.

To declare an iterator on a vector, use: vector¡type¿::iterator. There are two methods in vector to return the beginning and end of the list:

```
vector<int> a;
a.push_back(3);
a.push_back(1);
a.push_back(4); // a now contains 3,1,4
vector<int>::iterator i = a.begin(); // i now points to the
   beginning of the list.
cout << *i << ' '; // print out the value of the element pointed
   to by i (3)
++i; // move forward to the next element
cout << *i << ' '; // priint the current element (1)

for (vector<int>::iterator j = a.begin(); j != a.end(); ++j) //
   print the whole list
  cout << *j << ' '; // print each element
```

It is not always possible to go backwards, but in the case of vector, you can start at the end and work backwards:

```
for vector<int>::reverse_iterator i = a.rbegin(); rit!= myvector.
   rend(); ++rit)
  cout << *i << ' ';
```

It is also possible to modify the list with an iterator by writing to the iterator. The following code adds one to every element in the list:

```
for (vector<int>::iterator i = a.begin(); i != a.end(); ++i)
  ++*i;
```

Because of this, it is illegal to attach an iterator to a constant object (because it can change the object). For objects that are not supposed to change, there is a const_iterator that only supports looking (not changing) the value.

```
const vector<double> mylist = {1.5, 2.5, 3.5}; // list cannot
   change now.
for (vector<double>::const_iterator i = a.begin(); i != a.end();
   ++i)
  cout << *i;
```

map

The map class is designed to store a mapping of key to value (a dictionary). Very often, we need to lookup one value (the key) and find the corresponding value. For example, in the stock market, stocks are listed by symbol. In order to look up the price of AAPL (Apple Computer), we need a data structure that rapidly finds the record for AAPL. Since there are thousands of stocks, looking through all of them for AAPL is not an attractive option.

The map class is implemented using an RB-Tree, something studied in data structures classes. Here we are just concerned with using the map class to achieve a goal.

To declare a map, decide what kind of value to use for the key and value. For example, for stocks we might look up a string and get a corresponding price which is a double. This would be declared as:

```
map<string, double> stocks;
```

There are three common operations needed with maps. First, we must be able to add new values into the map (or replace a value already in it). Second, we must be able to look up a key and find the corresponding value (if any). Finally, sometimes we want to iterate through the entire map. These three operations are shown below:

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

int main() {
  map<string, double> stocks;
  stocks["AAPL"] = 162; // this is one way to add a key, value
     pair

     // this is a second way to insert a new value that only
        works in C++11
  stocks.insert({"IBM", 108}); // another way to add a key,value
     pair


}
```

unordered_map

A map is pretty fast for looking up a single element, but the STL unordered_map is even faster. This class uses a hashmap to turn the key into the location where it may be found. It is not sorted, however, so while iterating through a map will traverse in sorted order, the traversal for an unordered_map will be random and can even change over time.

How References work

This section requires a knowledge of pointers. If you don't know pointers, read up on them first, then come back

Really, references ARE pointers. But C++ wont admit it. References are safer (not safe)

No pointer arithmetic.

References must be initialized

```
int main() {
  int a = 2;
  int& b; // error, b must be initialized
  int& b = a; // this is ok.  b is now an alias for a

  b = 3; // a is now 3 // b syntactically looks like the thing it
      points at.
// I know b is really a pointer, so   .
  a = 4; // both b and a are now 4
  cout << b <<    \n   ; // prints the value of a
  cout << &b <<    \n   ; // prints the address of a
  cout << sizeof(b) <<    \n   ; // prints the size of a!!!
}
```

The reference b is locked to a (cant change). This is analogous to the pointer:

```
int* const p = & a; // p is a pointer that can only point to a
```

except that with the pointer, casting could break it:

```
((int*)p)=0; // set pointer to null?  Probably compiler will stop
   me,
*(int**)&p = 0; // p now points to nothing.
```

Classes and Objects in C++

A class in C++ is a specification of an object. To create an object, we say we are instantiating or creating an instance. Classes are a single, big declaration, and must end in a semicolon. The following is an empty class definition.

```
class A {
};
```

All outer level classes are public, no need to say public. Anything inside a class is private by default and cannot be seen outside the class. The dot operator is used to look at symbols within a class.

```

class A {
  int x; // no declaration, so it's private
private:
  int y; // y is also private, so x and y cannot be seen by the
     outside world
public:
  int z; // z can be seen by everyone
};

A a1; // create an object of type A
a1.z = 5;
```

```
a1.x = 3; // illegal, private variables are not accessible
   outside the class
a1.y = 4; // illegal, private variables are not accessible
   outside the class
```

private, protected, public are sections followed by colons

```
int x; // declaration, ends in a semicolon

class A {
```

```
private:
  int x, y;
public:
  void f();
  void g();
}; // declaration, ends in a semicolon
```

The three ways to create an object (instantiate a class):

```
int main() {
  A a1; // declare a variable of type A.  a1 contains x and y
  A* p = new A(); // create an object of type A on the heap and
     return a pointer
  A(); // calling the constructor creates an unnamed, temporary
     object which disappears by the semicolon
}
```

Example:

```
class Fraction {
private:
  int num, den;
public:
  Fraction(int n, int d) { num = n; den = d; } // methods in the
     class are automatically inline
  Fraction add(Fraction right) {
    Fraction ans(                    ,                         );
    return ans;
  }
};
```

```
class Fraction {
private:
  int num, den;
public:
  Fraction(int n, int d) { num = n; den = d; } // methods in the
     class are automatically inline
  Fraction add(Fraction right) {
    return Fraction(                    ,                         );
  }
};
```

1. Implement the following class Fraction. Read in numerators and denominators in the following format and write the code to print out the sums

```
2
1 2    1 3          should print out 5/6
1 2    1 2          should print out 1/1
```

```
class Fraction {
private:
  int num, den;
public:
  Fraction(int n, int d) { num = n; den = d; } // methods in
     the class are automatically inline
  Fraction add(Fraction right) ;
  Fraction sub(Fraction right) ;
  Fraction mult(Fraction right) ;
  void print() const ; // this method is READONLY.  It
     promises not to change the object
};
```

Questions

1. What is the relationship between a class and an object?
2. How do you create a object of type Fraction?

Exercises

1. Add gcd function and divide both numerator and denominator by gcd to make sure every fraction is always in simplified form. For example, for the fraction 2 / 4, compute the gcd(2,4) which is 2. Divide both numerator and denominator by 2 to yield (1 / 2) which is the simplified fraction.
   This highlights an important object-oriented principle: never let bad state in the object. In this case, never let the object hold an unsimplified fraction, fix it as it is being built.
2. Design a class Vector2D or Vector3D. What kinds of operations must be supported for vectors? Designing this problem on your own as opposed to implementing what you are told is an important learning exercise.

Constructors and Destructors

Constructors

The constructor in C++ is a special function with the same name as the class. Constructors are automatically called when objects are created to ensure that objects are correctly initialized.

Rules for constructors:

- Constructors cannot have a return value, not even void.
- If no constructor is written, a default constructor that doe nothing.
- If any constructor is written, no default constructor is generated.

```
class A {
private:
  int x;
public:
};
// automatically generates default constructor:  A() {} above

A a1; // variables within a1 are uninitialized.
```

If a constructor is written, then the default constructor is *not* generated. The example below contains a constructor that takes a single integer parameter. It is therefore illegal to declare variable a1 with no parameters since a constructor was defined with an integer parameter.

```
class A {
private:
  int x;
public:
  A(int xin) { x = xin; }
};

```

```
A a1; // this is illegal, no default constructor
A a2(5); // this is ok, calls the constructor defined in the
   class
```

Initializer Lists

Initialization of data members in a class can happen in two ways. First, you could assign values. For example:

```
class A {
private:
  int x;
public:
  A() {  x = 3; }
}

int main() {
  A a1;
}
```

However, if the value is constant, even the constructor is not allowed to assign a value to it. In this case, the programmer must use initializer list syntax.

```
class A {
private:
  const int x;
public:
  A() : x(3) { }
}

int main() {
  A a1;
}
```

Destructors

Destructors have the name of the class preceded by a tilde ( ). There is no reason to write a destructor unless something is allocated during the lifetime of the object that must be given back. Later, with dynamic memory, you will do exactly that. For now, the destructor can be used to display a message every time an object is going away.

```
class A {
private:
  int x;
public:
  A(int x) : x(x) { cout << "An object of type A is being born\n"
      ; }
  ~A() { cout << "An object of type A is dying\n"; }
}

void f() {
  A a1;
}

int main() {
  A a1;
  f();
}
```

const Members and const Methods

A data member is a variable declared within a class that is in each object created. If it is declared const, this means that it can never change for the life of the object. For example, the following class definition contains an id. Every time an object of type Person is created, an id is generated. This id cannot be changed for the life of the person.

```cpp
class Person {
private:
  const int id;
public:
  Person() : id(1) {}
};
```

Note that in this case, even if the constructor tries to assign id, it is illegal to do so. Once id is initialized, it can never be written again:

```cpp
  Person() { ud = 1; } // illegal, id is const cannot be assigned
```

A const method promises not to change the object. For example, in the following example, the method print() while not written in the greatest style, is written so it does not change the object.

```cpp
class Person {
private:
  const int id;
public:
  Person() : id(1) {}
  void print() const { cout << "Person, id=" << id; }
};
```

static Members

By default, symbols declared within a class refer to instances of that class. This means that a variable declared within a class exists within each object. A function (method) declared within the class can be called for each particular object. Putting the keyword **static** in front of a declaration means that it is shared by the entire class. For variables, that means there is only a single value no matter how many objects are created.

For example, the following class Zebra contains a declaration of a string name. This means that every Zebra object has a name. But the variable count is declared static, meaning that there is only one count shared by all Zebras.

Note that every occurrence of a static variable within a class turns into extern. Each variable must be defined outside the class. In the example below this is done immediately after the class definition.

```
class Zebra {
private:
  string name;
  static int count;
public:
  Zebra(const string& name) : name(name) {
    count++;
    cout << "Zebra born, count = " << count << '\n';
  }
  ~Zebra()  {
    count--;
    cout << "Zebra " << name << "dying, count = " << count << '\n
       ';
  }
  static int getCount() { return count; }
};

int Zebra::count = 0; // define the static counter

int main() {
  Zebra z1("Fred"); // created zebra named Fred, now there is 1
  Zebra Z2("Alice"); // create zebra named Alice, now there are 2
} // at the end of the program the destructors are called
```

Operator Overloading

C++ allows the programmer to overload existing operators that apply to new data types. This makes mathematical code look very natural.

C++ does not permit

1. Redefining operators for built-in types like int, float, double that already have operators. Operators cannot be changed once defined.
2. Defining new operators that did not exist, for example '**'

The simplest syntax for C++ operators is to consider operators outside of classes. The syntax simply replaces the name of a function with the keyword operator following by the operator. A binary operator takes two parameters and a unary operator takes one. For example:

```
Fraction operator +(Fraction left, Fraction right);
Fraction operator -(Fraction op);
```

would declare that when the compiler sees two fractions with a + operator between them, it should call operator +, and when it sees a fraction preceded by unary -, it should call operator -. The following shows this in code:

```
Fraction f1;
Fraction f2;

Fraction f3 = f1 + f2;
Fraction f4 = -f1;
```

The problem is that even though the syntax is simple, these functions will not be allowed to use the internal values from the fractions (num and den) which are needed to compute the results. Either they will have to use access methods getNum() and getDen() or we must declare them somehow special and allowed to access private components of the class. The second method is preferred, and inside the class, the operators are declared *friend* allowing the function outside to access the private symbols.

```
class Fraction {
private:
  int num, den;
public:
  Fraction(int n, int d) : num(n), den(d) {}
  friend Fraction operator +(const Fraction& left, const Fraction
     & right);
  friend Fraction operator -(const Fraction& a);
};

Fraction operator +(const Fraction& left, const Fraction& right)
   {
  return Fraction(left.num*right.den + right.num*left.den,
                  left.den*right.den);
}

Fraction operator -(const Fraction& a) {
  return Fraction(-a.num, a.den);
}
```

Questions

1. Is it possible to redefine 2 + 3 so that instead of addition, it subtracts?
2. Can we define a new operator +++ that adds 3 to a variable?
3. What is the precedence of a + b for fractions?
4. A class Fraction is defined. Write the declaration to define an operator * to multiply two Fraction objects.
5. A class Fraction is defined. Write the declaration of a unary - which negates the fraction.

Exercises

1. Write class Fraction and make the following main work:

```
int main() {
  Fraction f1(1,2);
  Fraction f2(1,3);
  Fraction f3 = f1 + f2; // should be 5/6
  f3.print();
  Fraction f4 = f1 * f2; // should be 1/6
  f4.print();
  Fraction f5 = -f4;          // should be -1/6
  f5.print();
}
```

Member Operators vs. Friends

C++ provides an alternate way of writing operators, as member functions. Member functions are always applied to an object of the type of this class. The parameter is called this. There is no first parameter. Therefore, a binary operator will only mention the right parameter, because the left is this. A unary operator will have no parameters, because the only parameter is this.

```
class Fraction {
private:
  int num, den;
public:
  Fraction(int n, int d) : num(n), den(d) {}
  Fraction operator +(const Fraction& right);
  Fraction operator -();
};

Fraction Fraction::operator +(const Fraction& right) {
  return Fraction(left.num*right.den + right.num*left.den,
                  left.den*right.den);
}

Fraction Fraction::operator -() {
  return Fraction(-a.num, a.den);
}
```

Separate Compilation

A small program can be written in a singe file. The C++ compiler expects this kind of program to have a suffix of .cc or .cpp or .cxx to show that it is C++.

Bigger programs must be split into multiple files. This way, many programmers can work on the project at the same time without colliding with each other.

In order to split an object-oriented program, the classes are defined in *header files*. These files by convention end in .h or .hh or .hxx but the C++ compiler will work with any name. This is the #include directive that you have used from the very first program in C++. In fact, the libraries themselves do not use any extensions at all – for example iostream or string does not end in .h.

Now, in order to split the code, you are about to write your own header files. The following example shows how a class Fraction, written in one C++ file, would be split into three files. First, there is the header file Fraction.hh which contains the definition of the class, but not the code. Then, the code is written in Fraction.cc. Since the code is no longer inside the class, all the members of the class are decared with Fraction:: in front of the symbols, to show that they are part of the Fraction class. Finally, the code that uses the class is put into test.cc

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

class Fraction {
private:
  int num, den;
public:
  Fraction(int n, int d = 1) : num(n), den(d) {}
  friend Fraction operator +(const Fraction& left, const Fraction
    & right) {
    return Fraction(left.num*right.den + right.num*left.den, left
       .den*right.den);
  }
};
int main() {
  Fraction a(1,2);
  Fraction b(1,3);
  Fraction c = a + b;
}
```

The first step is to define the class in a header file.

```
#ifndef FRACTION_HH_
#define FRACTION_HH_

#include <iostream>

class Fraction {
private:
  int num, den;
public:
  Fraction(int n, int d = 1);
  friend Fraction operator +(const Fraction& left, const Fraction
     & right);
};

#endif
```

Notice that the #ifndef, #define and #endif preprocessor directives are used to cut out the code if it is ever included twice. More on this later.

The main program (in main.cc) must #include the header file in order to define the class Fraction.

```
#include "Fraction.hh"
int main() {
  Fraction a(1,2);
  Fraction b(1,3);
  Fraction c = a + b;
}
```

Finally, the program must define the functions of class Fraction. This is done only once in Fraction.cc. No matter how many times in the program Fraction.hh is included, there is only a single copy of the functions.

```
#include "Fraction.hh"

// read :: as inside class Fraction , the constructor Fraction
Fraction::Fraction(int n, int d) : num(n), den(d) {
}

Fraction operator +(const Fraction& left, const Fraction& right)
   {
```

```
  return Fraction(left.num*right.den + right.num*left.den,
                  left.den*right.den);
}
```

Preprocessor Directives

What is the Preprocessor?

The preprocessor is a separate language that runs before the C++ compiler analyzes code. Any line beginning in # invokes the preprocessor.

The preprocessor can modify the program in a number of ways:

- Remove comments so the compiler does not consider them
- Include a file whose contents is injected
- conditionally take some action based on the value of a symbol
- Define string macros which are replaced by their contents

The first three actions are most useful, while the third is now largely obsolete in C++.

When the preprocessor detects a comment, it turns it into a single space. Thus if your program looks like the following:

```
int x/*this is a comment*/=1;
```

the preprocessor turns it into this:

```
int x =1;
```

before the compiler looks at the program. To bring in a file, use the #include directive. We have always done this without discussing what it does.

The third major contribution is that the preprocessor can conditionally compile code. One use is in temporarily disabling some code without cutting it completely:

```
int main() {
  f();
#if 0 // everything from here to #endif will turn into a single
   space
  g(); // temporarily remove this call while we work on something
      ...
  h(); // this is all gone...
#endif
}
```

The ability of the preprocessor to conditionally eliminate code is useful in a number of cases. First, it is illegal to define the same symbol twice. For example:

```
int x;
int x;
class A {

};
class A {

};
```

is illegal because it defines x and A twice. But this can easily happen if these declarations are in a header file. Even though the programmer can make sure never to write:

```
#include "A.hh"
#include "A.hh"
```

it is impossible to stop the following situation:

```
//file B.hh
#include "A.hh"
class B {

};
```

```
//file C.hh
#include "A.hh"
class C {

};
```

In the above case, a programmer wanting to define both B and C must write:

```
#include "B.hh"
#include "C.hh"
```

yet this will bring in the definitions in A.hh twice. The solution is to use the preprocessor in file A.hh

```
#ifndef A_HH_ // the first time, A_HH_ is NOT DEFINED, so this is
    true
#define A_HH_ // now the symbol is defined, the 2nd time it will
   be FALSE
class A {
  ...
};
#endif
```

The above is the traditional (and still standard) technique to avoid this problem. Today, there is a new, non-standard technique that is more convenient but not yet supported by all compilers. Still, it is supported by g++ and clang, and will presumably become part of the standard soon.

```
#pragma once
\end{lstlsitng}

automatically prevents the file from being included twice without
    having to invnet a name.

\section{Macros: Mostly Obsolete}
The last use of the preprocessor is in defining macros.  This is
   largely obsolete due to new features of C++ like inline
   functions and templates that provide the same capabilities in
   much more sophisticated and less error-prone ways.

The preprocessor allows defining a symbol:
\cpp
#define A blah
```

From that point on, every time the preprocessor encounters the symbol A it replaces it with blah. Note this is not every occurrence of the letter A. For example:

```
int A = 5;
int THISISATEST = 6;
```

becomes

```
int blah = 5;
int THISISATEST = 6;
```

Notice that the A embedded in THISISATEST does not change.

Occasionally it can be so hard to figure out what the preprocessor is doing that you want to see the results. Every compiler has a way of saving the output of the preprocessor. On g++ and clang++ for example, it is:

```
g++ -E myfile.cc
```

which does not compile but shows the preprocessor output that would go into the compiler.

Questions

1. When does the preprocessor run, and where does its output go?
2. What does #include do?

Exercises

1. Write a header file for class Fraction so that it can be included twice and still does not give errors.
2. What is wrong with the following code?

```
#ifndef Fraction
#define Fraction
class Fraction {
private:
  int num, den;
};
#endif
```

Using the Optimizer

The optimizer is a powerful way to make programs faster. In C and C++, an optimizer is allowed to rewrite code as long as a correct program orks the same way with and without optimization. Note that this means if you write an incorrect program behavior could, in fact, change. Also, if the compiler itself has a bug (unlikely, but it happens) then optimizing coul change behavior of a correct program.

Consider the following program to count to 10 billion (10^10). This is a lot of operations even for a computer, so it will take some time.

```
int main() {
  for (long long  i = 0; i < 10000000000L; i++)
    ;
  return 0;
}
```

On my laptop, when I use the time command in cygwin to measure how long the execution takes, it shows:

```
$ time ./a.exe

real    0m31.785s
user    0m31.750s
```

You will notice that the above program does nothing, so an optimizer, if it recognizes that fact, is free to get rid of the loop since it has no effect on the program (other than the time it takes). When compiled with the option:

g++ -O3 opt.cc

the time goes down to essentially zero:

```
$ time ./a.exe

real    0m0.019s
user    0m0.000s
sys     0m0.015s
```

In fact, increasing the loop count makes no difference, because the compiler is eliminating the loop.

Suppose the program is changed to sum the numbers from 1 to 10 billion. Now there is a purpose to the loop:

```
int main() {
  long long sum = 0;
  for (long long  i = 0; i < 10000000000L; i++)
    sum += i;
  return 0;
}
```

Yet when compiled with optimization, this program too runs instantly. The reason is that the program does nothing with the sum, and therefore the loop is again optimized away. However, if you print the result, then the program is affected by the loop, and it stays:

```
#include <iostream>
using namespace std;
int main() {
  long long sum = 0;
    for (long long  i = 0; i < 10000000000L; i++)
    sum += i;
  cout << sum << '\n';
  return 0;
}
```

Write a program to read in a number and determine all the primes up to that number. For example, if you enter the number 10, the answer should be 2 3 5 7.

It will be a little inefficient to print all the numbers, so to avoid ruining your timings, once you have the program working just add all the primes up and just print out the sum of all the primes. For example, the sum of all the primes below 10 is: 2+3+5+7 = 17.

Try running your program both with and without optimization. How much faster is it with optimization?

The Make Utility

Building Larger Programs

In order to build programs with multiple files efficiently, it is vital to avoid recompiling everything every time a change is made. Most small changes only affect a small part of the code.

The make utility in Unix is often used, on all platforms to build programs. It is old, but it works well. This guide is a quick introdution to make, and to build tools in general.

The make command by default reads a file in the current directory named Makefile. This file contains a list of rules to build targets. Typing make will attempt to build the first target. That target may in turn depend on other targets and trigger more rules.

The simplest Makefille is something like:

```
myprog: myprog.cc
	g++ myprog.cc -o myprog
```

The first line defines the dependency, that myprog depends on myprog.cc. The make program checks the timestamp on both files, and if myprog.cc iis newer than myprog, it executes the second line (the command) to build the program. The g++ command is indented with a tab, which is required. When executed, g++ compiles the program and generates the executable myprog (or on Windows, myprog.exe).

To build a program comosed of test.cc Fraction.h and Fraction.cc, the following makefile would work. This time, the program fraction depends on two .o files, which in turn depend on the .cc files and the .h file.

```
fraction: test.o Fraction.o
	g++ test.o Fraction.o -o fraction

Fraction.o: Fraction.cc Fraction.hh
	g++ -c $<

test.o: test.cc
	g++ -c $<
```

Handling Build Options

There are two options that are particularly important to programmers. The first is turning on debugging information, wiithout which the debugger will not function properly. Debugging information is added into compilation using the $-g$ option on g++. The second is optimization which makes the code much faster and more efficient. In some rare cases, optimized code can go 10 times faster. Moreover, using it can reduce use of memory, and since memory is shared by mutiplpe cores, there are algorithms that will be far more parallelizable if the code is optimized.

To compile with optimization, use the -O2 or -O3 option for g++. You could, of course, leave optimization on all the time, but it makes debugging the program more difficult. Similarly, if the program is full of debugging information it can be a lot bigger and slower to load. Therefore, we build these options into the Makefile so that a single change can rebuild the whole program using different options.

The following Makefile contains some of these standard features. The variable CPP is set to the current compiler being used. The variable DEBUG can be set to -g for debugging, or commented out #-g for no debugging. The variable OPT can be set to -O3 for the fastest code, or commented out for no optimization.

Last the target clean will delete the whole program if desired. Use something like this in your builds, and particularly, in your group project.

```
CPP=g++
DEBUG=#-g   # debugging is off for now, remove # before -g to enable
OPT=-O3

fraction: test.o Fraction.o
	$(CPP) $(DEBUG) test.o Fraction.o -o fraction

Fraction.o: Fraction.cc Fraction.hh
	$(CPP) $(DEBUG) $(OPT) -c

clean:
	rm *.o fraction          #remove all the binary files, clean up
```

The problem with Make is that it takes a lot of your work to figure out how to build a makefile properly, and every time the structure of the program changes, the makefile must be changed as well. Additionally, if you want to turn on debugging, the makefile has no dependency built in to realize that you just changed every command from:

```
g++
```

to

```
g++ -g
```

In order to do so, you would have to manually force a rebuild by using make clean followed by make. There are variants of make such as gnu make that have commands designed to identify when the commands change and rebuild the makefile. These are more complicated and beyond the scope of this short guide. But rather than using a bandaid which can make the make utility a bit easier to use, the next level is to consider another tool that can automatically configure a Makefile so that you do not have to generate one. This would be the cmake utility.

cmake

The cmake utility is a meta-program that builds code. But instead of building it directly, like make, its job is to have a set of commands that define what your program is, and a set of back-end tools (like make) that actuall build the code. Then, CMake automatically generates the more complicated commands that the back end tool like make uses.

You can use cmake to automatically maintain a makefile that builds your program. But you are not limited to make. Today, there are faster, more modern tools such as ninja that are much faster than make. The cmake tool is designed to use many different build tools, and is configurable.

The CLion IDE uses cmake to build code, and so for simple programs you do not even have to think about it. However, for more complicated applications you will have to know how to modify the script controlling the build, so you cannot simply rely on the IDE to take care of everything for you.

In order to use cmake, you will first need to install it. Search for how to install cmake on your system.

The cmake utility uses a file called CMakeLists.txt. This file must, at minimum list which files are needed to build, and what is being built.

The following CMakeLists.txt shows a simple CLion project. The first line specifies which version of cmake is required. The second specifies the name of the project. The third defines a set of rules using c++ 11, which probably specifies a number of things, including which c++ compiler is being used, what version of the standard (c++11). No doubt this results in a command

```
g++ -std=c++11
```

The next to last line sets the list of all files needed to build the code. If a second file is added to the project, all that is necessary is to add it to SOURCE_FILES.

Notice that there is nothing special about the variable SOURCE_FILES, it could be called anything. It is used in the following command which defines how to build the executable. This last line defines that in order to build session01, use all the files in ${SOURCE_FILES} .

```
cmake_minimum_required(VERSION 3.8)
```

```
project(session01)

set(CMAKE_CXX_STANDARD 11)

set(SOURCE_FILES 08summation.cc)
add_executable(session01 ${SOURCE_FILES})
```

For more complicated projects, it is possible to specify multiple executables, multiple source files, building libraries, and many other options. The most important thing is that while a tool like make requires you to identify which files depend on other files, cmake is responsible to automatically figure out that list and generate the right makefile. So cmake does not actually build your program. Typically, it just generates the underlying makefile which we do not even see.

Dynamic Memory

Old C Memory Allocation

In C, memory can be allocated dynamically (from the heap) using malloc and returned using free(). In C++ these still work, but the new way of allocating memory is to use the operators new and delete. Both systems may be used in the same program, but may not mix and match.

For every pointer allocated with malloc, the memory must be returned to the system with free. The same is true of new and delete.

```
int* p = (int*)malloc(1000 * sizeof(int)); // allocate enough for
   1000 integers
...
free(p); // give back the memory so it can be reused.
```

Memory must be returned, or it is ”leaked.” If leaked, it will be recovered when the task ends, but until then the memory is not available for use. Tasks that leak a lot of memory can impact performance of the computer, and may eventually crash.

The ”New” Way

In C++, memory is allocated using operators **new** and **delete**. Both C and C++ memory operations can coexist in the same program, but memory allocated with malloc cannot be returned with delete, and memory allocated with new must not be returned with free().

```
int* p = new int; // new C++ way, allocate a single number
   dynamically
...
delete p;
```

If a block of numbers is allocated, it can be done using new[] and it must be deallocated using delete[] (with square brackets).

```
int* p = new int[1000]; // new C++ way
...
delete [] p;
```

It is illegal to:

- Allocate memory with new[] and delete it without the square brackets
- Delete memory that has not been allocated dynamically.
- Delete the same memory more than once.
- Not deallocate memory that was allocated with new.

Examples:

```
int a;
delete &a; // error, a was not declared with new
int* p = &a;
delete p; // error, p is not pointing to dynamically allocated
   memory
int* q = new int[1000];
delete q; // error, if allocated using new [], use delete[]
int* r = new int[100]; // error, this memory leaks because r is
   never deleted
int* s = new int[100];
s++;
delete[] s; // error, the pointer deleted must be the same as the
   one returned by new.  s has been changed to s+1.
```

Questions

1. malloc and free are obsolete, the ”old” way of allocating memory. What is the replacement?
2. The memory pointed to by p is returned with **delete** p;. How must the memory have been allocated earlier?
3. The memory pointer to by p is returned with **delete** [] p;. How must the memory have been allocated earlier?

Exercises

1. Write a class that has a constructor with a single integer specifying the size. Allocate that many characters and set them all to 'a'. Then in the destructor, make sure you give the memory back.
2. Write a class to represent a matrix with rows and columns. The class should have the following data:

   ```
   class Matrix {
   private:
     int rows, cols;
     double* m;
   public:
   };
   ```

   Write the constructor and destructor to correctly allocate the memory. Make sure the values in the matrix are all initialized with 0.

Copy Semantics and Dynamic Memory

In Java, objects cannot be copied by default. If you want to copy, you must implements Cloneable.

```java
public class Vector3D implements Cloneable {
}
...
Vector3D v1 = new Vector3D(1,2,3);
Vector3D v2 = v1; // not a copy, v2 points to same object
v2 = v1.clone(); // copy
```

In C++, copying objects works by default. A new object can be created that is an exact copy of an existing one.

```cpp
int main() {
  Vector3D v1(1,2,3);
  Vector3D v2 = v1; // v2 is a copy of v1.  This is NOT the
     operator =
  Vector3D v3(v1); // alternate constructor syntax.
}
```

The default copy constructor copies each member bitwise. No guarantee is made about spaces between the data members. Example:

```cpp
class A {
private:
  char x; // one byte, aligned to 8 bytes on a typical memory
     architecture
  int y; // your compiler will probably skip 3 bytes for speed
}
```

On a typical 64 bit machine the above is 8 bytes. The int is aligned to byte 4.

Copy Semantics

A class that allocates a resource (such as memory) in the constructor must give it back. To guarantee that the programmer does not forget, a destructor method is automatically called when the objects is about to die. The constructor has the same name as the class; the destructor has the same name with a tilde before it.

```
class A {
private:
  int* p;
public:
  A() {
    p = new int[5];
  }

  ~A() {
     delete[] p;
  }
};
```

The destructor does not need to be written unless the constructor allocates a resource that must be returned. If the constructor and destructor are written, then the problem is that C++ allows objects to be copied, and this creates a problem.

Instead, we override the default copy constructor and create one that correctly copies the logical object. There are two contexts where C++ copies an object. The first is when the programmer explicitly creates a new object that is a copy of the old:

```
  A a1;
  A a2 = a1; // a2 is a new object that is an exact copy of a1
  A a3(a1); // alternate syntax: a3 is also an exact copy of a1
```

The other circumstance is when a copy is made in order to pass by value into or out of a function:

```
void f(A a) { ... }
A g() { return A(); }
```

```
int main() {
  A a1;
  f(a1); // a1 is copied into the formal parameter a
  A a2 = g(); // the return value of g is copied into a2
}
```

As long as only values are in the object, it works. The following code creates a copy that does this correctly

```
A(const A& orig){

}
```

There is also

```
class A {
private:

public:
     A() {      }
     ~A() {      } //destructor
     A(const A& orig){      } // copy constructor
     A& operator =(const A& orig) {    }  // copy an object once
        it already exists
};


Vector3D v1(1,2,3);
Vector3D v2 = v1; // calls copy constructor

Fraction
     2
     1 / 2     1 / 3               print out f1 + f2, f1 * f2
     1 / 2      1 / 2              print out 1/1  (simplify using
         gcd)

Vector3D (x,y,z)

Make it work with operator overloading

Vector3D a(1,2,3);
Vector3D b(2,1,1);
Vector3D c = a + b; // use overloaded operator +

class Vector3D {
```

```
private:
  double x,y,z;
public:
  Vector3D(double x... ) {  this->x = x;  }

// this is the  java way  (looks like it), but there are
   better ways in c++
  Vector3D add(Vector3D b ) {

  }
// first, pass by reference, not by value (this is internaly the
   way java works)
  Vector3D add(Vector3D& b ) {

  }

// The above could change the value in b, so pass as a const
  Vector3D add(const Vector3D& b ) {

  }

// The above could change the value in b, so pass as a const
  Vector3D operator +(const Vector3D& b ) {

  }
// readonly method. cannot change the object..
  double abs() const {     . }
  Vector3D unit() const { .. .}
  void print() const {  // print out the vector
  }
};

v1.add(v2)

v1 + v2
v1.print();

cout <<   hello     << 5 + 4 << ...
Vector3D:
     2
     1 2 3          2.0 3.5 1.5          output    1 5.5 4.5
                              output    magnitude of (1,5,5, 4.5)
                                   sqrt(squares )
                              output normalized vector  (1/m,
                                 5.5/m, 4.5/m)
     0 1 -3              ..              output    sum
                              output    magnitude
                              output normalized vector


class Fraction {
private:
  int num, den;
public:
  Fraction(int n) { num = n; den = 1; }
  Fraction operator + (const Fraction& right) {... }
};

Fraction f1(2);
Fraction f2 = 2; // both call constructor
Fraction f3 = f1 + 1;
     // automatically calls
     f3 = f1 +Fraction(1);  // calling the constructor creates a
        temp of type Fraction
Fraction f3 = 1 +f1 ; // this will not work, left hand side must
   be a fraction!

So instead, use FRIEND notation...
class Fraction {
private:
  int num, den;
public:
  Fraction(int n) { num = n; den = 1; }
  Fraction operator + (const Fraction& right) {... }
  friend int f(Fraction x);
  friend Fraction operator + (const Fraction& a, const Fraction&
     b);

  friend std::ostream& operator <<(std::ostream& s, const
     Fraction& f) {
     s << f.num <<     /     << f.den;
  return s;

}
// more compact would be...
  friend std::ostream& operator <<(std::ostream& s, const
     Fraction& f) {
     return s << f.num <<     /     << f.den;
}

};


cout << f1 <<           << f2;
```

```
 int f(Fraction x) {
  x.num = 1; // I am a friend, I have full rights just like a
     method declared within the class.
}

Fraction operator +(const Fraction& a, const Fraction& b) {
  Fraction ans;
  ans.num = ..
  ans.den =
  return ans;
}

// more compact way of writing it
Fraction operator +(const Fraction& a, const Fraction& b) {
  Fraction ans(                               ,
                                             )
  return ans;
}

// even more compact way of writing it
Fraction operator +(const Fraction& a, const Fraction& b) {
  return Fraction(                               ,
                                             );
}
```

Pointer Ambiguity in C++

NULL is the old way of writing a pointer that does not point to anything. #define NULL ((void*)0L)

In C++, Stroustrup added the constant 0 as nul

The problem is 0 is also an integer.

```cpp
void f(int x) {

}
void f(char* p0) {

}


int a = 0;
char* p = 0;
f(a); // not ambiguous
f(p); // not ambiguous
f(0); // ambiiguous
```

New C++11 way of writing null:

```cpp
f(nullptr); // not ambiguous
```

C++ Operator Precedence

Type Ambiguity

If two conversions are defined for a class, there may be two alternate ways of evaluating an expression. For example, given that an operator + is defined fora class Fraction, and the class has a constructor that turns an integer into a Fraction, and an operator int() that converts the fraction back to an integer, a mixed-type expression with one integer and one Fraction could be handled two ways: either the integer is converted to a Fraction, or the Fraction is converted to an integer.

```
Fraction f1;
Fraction operator +(Fraction, Fraction)
Fraction(int);
operator int(Fraction);
```

The following expression is ambiguous and results in a compiler error. It is not clear whether to convert 1 to a fraction, or convert f1 to an integer.

```
f1 + 1
```

To resolve, manually convert. Either convert the integer to a Fraction or the Fraction to an integer.

```
Fraction f2 = f1 + Fraction(1)
int j = int(f1) + 1
```

C++ Classes, Containment, Initializer Lists

In C++, constructors can be written to assign values to data members.

```
class A {
private:
  int x;
public:
  A() { x = 0; }
};
```

However, there are circumstances when assignment will not work. In this case, with the variable declared const, it is not allowed to change.

```
class A {
private:
  const int x;
public:
  A() { x = 0; } // THIS IS ILLEGAL, x cannot be assigned!
};
```

In the above case, instead of writing the constructor to assign the value of x, we need to *initialize* the value of x. This uses a new syntax: the C++ initializer list.

```
  A() : x(0) {}
```

There are other reasons that a data member cannot be assigned. It may be an object that must have an appropriate constructor call. The following example shows a class Car which contains four Tires and an Engine. Each object must be initialized. In order to initialize the Car, it is necessary to initialize the four Tires, and then then Engine.

```
class Tire {
private:
  int pressure;
public:
  Tire(int p) : pressure(p) { }
};

class Engine {
private:
  const int horsePower;
public:
  Engine(int hp) : horsePower(hp) { }
// this would be illegal
//  void setHorsePower(int hp) { horsePower = hp; }
};

class Vehicle {
private:
  int speed;
public:
  Vehicle(int speed) : this->speed(speed) {}
};

class Car : public Vehicle {
private:
  Tire t1, t2, t3, t4;
  Engine e;
public:
  // assume all four tires get the same pressure, how to specify
     constructor?
  Car(int speed, int pressure, int hp) : Vehicle(speed), t1(
     pressure), t2(pressure), t3(pressure), t4(pressure), e(hp)
 {  }
}

int main() {
  Tire t1(28);
  Car c1(55, 28, 150);
}
```

What would this look like in java??? (build complete examples in both languages)

```
public class Car {
private Tire t1, t2, t3, t4;
private Engine e;
public Car(int pressure, int hp) { t1 = new Tire(pressure) ...}
}
```

virtual Functions

In Java, classes by default allow polymorphism, which means each object must identify what class it belongs to.

This in turn means that every object contains a pointer (4 bytes). In C++ this overhead does not exist unless specfcially requested.

```
class Vehicle {
public:
  virtual void payToll();
}
```

The keyword virtual instructs the compiler to add information to this class, and all descendents. There is no more overhead for two virtual functions than for one.

Calling a virtual function through a pointer or reference is slower than a regular function.

A regular function is called directly A virtual function first looks up the object type, and calls the appropriate

```
Car c1(35);
c1.payToll(); // this calls Car::payToll,known at compile time,
   no time penalty
Vehicle* vp = &c1;
vp->payToll(); // this call must look at the tag at the front of
   the object and call the appropriate function.
```

Templates and Generic Programming

Object-oriented programming offers a way to save effort by reusing code. Specifically, new objects can be created that inherit from or contain existing objects, thus reducing effort. However, not all code can be simplified using object-oriented techniques. Another method called *generic programming* allows reduction of effort in different situations.

The situation is where there is the same logic, yet it must be applied to different objects not related by inheritance. Generic programming allows the programmer to specify an agorithm or class, and pass the type of the object to it as a parameter.

Templates are very powerful, but they work only at compile time.

Templated Functions: Sorting

Consider a sort algorithm. The following example shows a traditional bubblesort that can sort an array of integers. In order to sort a different array, the programmer would have to write an almost identica function with a different type list:

```
void bubbleSort(int x[], int n) {
  for (int i = 0; i < n-1; i++)
    for (int j  0; j < n-1-i; j++)
      if (xj] > x[j+1]) {
        int temp = x[j];
        x[j] = x[j+1];
        x[j+1] = temp;
      }
}
```

The only thing that keeps the above code speciific to integers is the two uses of the keyword int, one in the parameter (the array itsellf) and the other for the type of the variable temp which is used to swap the elements of x if they are out of order.

In order to make this function templated, a single template command precedes the function, defining a typename T. Then the two uses of int instead reference T

```
template<typename T>
void bubbleSort(T x[], int n) {
  for (int i = 0; i < n-1; i++)
    for (int j  0; j < n-1-i; j++)
      if (xj] > x[j+1]) {
        T temp = x[j];
        x[j] = x[j+1];
        x[j+1] = temp;
      }
}
```

No code has been generated at this point. But the moment a caller wants to use the function, one must be generated. If the programmer creates an integer array and passes it to the bubbleSort function, the compiler figures out that T must be int, a process called *type unification* . Then, the compiler generates a function specifically for int. If the programmer then creates an array of a different type (like string) the compiler generates another copy of the code, just for string. The following example shows each call to bubbelSort which look just like they are calling a function written specifically for them:

```
int main() {
  int a[] = {9. 8, 7, 5};
  bubbleSort(a, 4); // generates an integer sort function

  string b[] = {"this", "is" "a" test", "of", "strings"};
  bubbleSort(b, 6);
}
```

Templated Classes

Exception Handling

Multithreading

In C++11, multithreading became part of the language. Prior to that, each operating system had its own code, so threading was not portable.

To create a thread and start it running, include the thread header file, create a thread object passing it the name of a function to be executed in the background.

```
#include <thread>
void f() {
  for (int i = 0; i < 10; i++) {
    cout << "hello";
    sleep(1);
  }
}

void g() {
  for (int i = 0; i < 5; i++) {
    cout << "bye";
    sleep(1);
  }
}

int main() {
  thread t(f); // start executing function f in the background.
  g();
}
```

Questions

1. What is multithreading?

2. What is the parameter to the constructor when creating a thread?

Exercises

1. Write a program that computes the sum of the numbers from 1 to n. Pick a big n so it takes a few seconds on your computer. Can you split into two threads and compute half in each? How much time does it save?

Take the quiz

Mutexes

Computing in parallel works very well when the data being computed in each thread have nothing to do with each other. But when data is shared between threads, errors can be caused by two threads overwriting each other on the same data. This kind of error is called a *race condition*.

The following example shows a bank account where money can be deposited and withdrawn. To exaggerate the problem, we read the balance from the account, wait 1 second, then write back the modified amount. In reality, the interval between the computer reading the balance and writing back the modified value is measured in billionths of a second, but it does exist.

Suppose two threads A and B are reading the bank account, which is currently 100. Thread A deposits another 10. Thread B deposits 10. If both A and B read the current value at nearly the same time, then both believe it is 100. Suppose A gets in first. It adds 100 + 10 and computes the new balance which is 110. But B has already read 100, so it does the same thing. The result is that when B writes out its deposit, the balance is 100 + 10 = 110. In other words, two deposits have happened, yet only one has been recorded. This kind of error is called a race condition and can happen whenever two threads are accessing the same data in an uncontrolled manner.

To solve the problem, we create a mutex object which has two methods: lock and unlock. Whenever two threads are racing to be write to a value, one has to get in first. The mutex lock method will allow the first one to go through and make the second one wait until the first unlocks the mutex.

Here is the code which shows an exaggerated operation in which one second elapses between getting the old balance and storing the new one.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <unistd.h>
using namespace std;

class Account {
private:
    double balance;
public:
    Account() : balance(0) {}
    void deposit(double amt) {
        double newBalance = balance + amt;
        sleep(1); // wait 1 second to amplify the possibility
            of race condition
        balance = newBalance;
    }
    double getBalance() const { return balance; }
};

void backgroundDeposit(Account* a) {
    a->deposit(10);
}

int main() {
    Account a;
    thread t[10];
    for (int i = 0; i < 10; i++) {
        t[i] = thread(backgroundDeposit, &a);
    }
    cout << "Before waiting for all threads, balance = " << a.
        getBalance() << '\n';
    for (int i = 0; i < 10; i++)
        t[i].join();
    cout << "After waiting for all threads, balance = " << a.
        getBalance() << '\n';
}
```

In order to make the code work properly, a mutex object must **mut**uallly **ex**clude each thread so that only one at a time has access to the balance. The code shown below replaces the Account object, all the rest of the program remains the same.

```
class Account {
private:
    double balance;
    mutex m;
public:
    Account() : balance(0) {}
    void deposit(double amt) {
        m.lock();
        double newBalance = balance + amt;
        sleep(1); // wait 1 second to amplify the possibility
            of race condition
        balance = newBalance;
        m.unlock();
    }
    double getBalance() const { return balance; }
};
```

Accessing a mutex is quite slow compared to the operation in this case. As a comparison, the multithreaded version of this code running 10 threads, each doing 100,000 deposits which lock and unlock the mutex took 4 seconds with maximum optimization under g++. Without optimization, the single threaded version of this code (without mutex) runs in under 0.03 seconds. Obviously, multithreading is only worthwhile when the overhead of checking for race conditions does not exceed the time saved by running the program in parallel!

Questions

1. What is a race condition?
2. What is a mutex?
3. Is there any reason to create a mutex if there is only a single thread of execution?

Exercises

1. Extend the account class from this section to support deposit and withdraw.
2. Write a class that allows two threads to compute prime numbers. Since mutexes take time to use, try to minimize the overhead. How can you best optimize multithreaded performance to compute all the prime numbers up to 2 billion? How does the time compare between 1, 2, 3, and 4 threads?

Take the quiz

Regular Expression Matcher

Regular Expressions

Regular expressions are patterns that can match strings. Using regular expressions is a way to search for very complicated strings without all the complexity. Still, regex is a topic (and a mini-language) in itself. In order to learn regular expressions interactively, go to a website like regexr.com

The basic rules of regular expressions are fairly simple. Here is a table of the simple rules:

abc	a followed by b followed by c
`[ace]`	one of the letters a c or e
`[a-z]`	one of the letters a through z
`[^aeiou]`	any letter that is not a e i o or u
.	any single letter that is not a newline
ca?b	c followed by zero or one (optional) a followed by b
cab	c followed by zero or more a followed by b
ca+b	c followed by one or more a followed by b

There are other patterns, but even these simple rules can get far more complicated when combined. For example:

```
[0 -9]{3}[\(\.\ -][0 -9]{3}[\(\.\ -][0 -9]{4}
```

matches phone numbers of the form: 212.866.3322, (212)866-3322 or 212-866-3322

In C++, the regex library is now built into C++11. Just include the ¡regex¿ header.

The C++11 Random Number Generator

Random Numbers

Random numbers are a surprisingly hard subject. Most random numbers on computer are actually ”pseudo-random” meaning the computer generates a sequence of numbers that looks random, but actually is deterministic if you understand the algorithm.

C traditionally supplied the rand() function which returned an integer. Unfortunately there is no standard on the algorithm used, and it often is not very good. Seeding the random number from the time with srand does not help. This kind of random number can be used in casual games where the answer is unimportant.

It is much better to use the new C++ random number generators, which have well-defined behavior and very strong numerics.

```
#include <random>
#include <iostream>
using namespace std;
int main()
{
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(1, 6);

    for (int n=0; n<10; ++n)
        std::cout << dis(gen) << ' ';
    cout << '\n';
}
```

Questions

1. Why would a random number generator give the same numbers every time?
2. Why is it an advantage to test with a random number generator that starts the

same each time?

Exercises

1. Write a program to simulate pairs of dice rolls. Read in a number of trials, then count the distribution of each possible value 2 through 12.

See the following web page about randomness.

OpenMP

Modern computers have multiple cores. This means they can compute faster in parallel, provided a program is able to run more than one thread. But multithreaded programming is hard as we have seen. So there is a library designed to automatically parallelize code with minimal human interaction. This library and standard is called OpenMP.

To build a c++ with OpenMP, compile with an option. For g++:

```
g++ -fopenmp test1.cc -o test1
```

This will link to the OpenMP library. When the program is run, it will read the state of an environment variable. To control the number of threads in a Unix shell:

```
export OMP_NUM_THREADS=4
```

then run the program, which will launch that many parallel threads.

The following simple code runs 10 iterations on a loop. Each one prints hello with a number and sleeps for one second. This would mean that the program would take 10 seconds to run. But if you run with OpenMP using the directive #pragma parallel the four threads set above will each get one and print out at the same time. This means that the order of completion is not guaranteed, so the output is a bit messy. It is still going to be necessary to be careful lest the threads corrupt some data. The threads must have their own private variables to work with, and typically share an array. The next section will show a simple prime number solver running in parallel.

```
#include <iostream>
#include <unistd.h>
using namespace std;

void f() {
#pragma omp parallel for
    for (int i = 0; i < 10; ++i) {
        cout << "hello" << i << '\n';
        sleep(1); // sleep for one second
    }
}
int main(int argc, char* argv[]) {
    f();
}
```

The above code *does not run on Msys under Windows*. It is tested only under Linux. Windows is capable of running OpenMP, but the threads in a Linux shell are not properly running. Either run a compiler native to windows (like Visual C++) or install Cygwin which has a more complete set of unix libraries.

The last thing to consider: memory bandwidth is limited. Each processor can read from local cache, but if multiple threads are writing to memory, processors will have to wait. One process is typically able to go at nearly 100% efficiency, barely waiting for memory if code is well written. With a second processor, there will be times when each processor is waiting for the other which is using the memory, so two processors typically do not go 100% faster than one. The second gets perhaps 80%. The next two get even less, so that running 4 processors, the result is often no more than 3 times faster, and may be far less.

Parallel Prime Number Search

In order to write a prime number solver that works in parallel, pick a loop that takes a long time to run, and add a directive in front of it instructing the OpenMP system to split it among the threads. The simplest directive to split is

```
#pragma omp parallel for
```

which tells the threads to take turns picking an iteration from the loop. This first approach is not very efficient because it takes time to start up threads and to assign them work, so it is most efficient to have a big unit of work for them to do. And whenever the threads are accessing common data, mutexes must be generated to prevent them from accessing the same data. In this case, when looking at a given number, the thread could immediately find that it is not prime because it is evenly divisible by 2, or 3, or the thread might have to go through all checks and discover that it is prime. This means that threads will finish at varying times and go back to get the next number to work on, wasting time.

In a second variant, we can try allocating a block of numbers for each thread to work on, but that is more complicated.

Self Check: Fundamental C++ Skills Mastery

See whether you understand what each of the following code snippets do. This section is about operators and statements

Loops and Logic

```
for (int i = 0; i < 10; i++)
  cout << i << ' ';
cout << '\n';
```

```
for (int i = 10; i > 0; i--)
  cout << i << ' ';
cout << '\n';
```

```
for (int i = 10; i >= 0; i--)
  cout << i << ' ';
cout << '\n';
```

```
for (int i = 10; i > 0; i++)
  cout << i << ' ';
cout << '\n';
```

```
for (int i = 1; i <= 3; i++) {
  for (int j = 1; j <= 3; j++)
    cout << i+j << ' ';
  cout << '\n';
}
```

```
int n = 19;
while (n > 0) {
  cout << n << ' ';
  n /= 2;
}
cout << '\n';
```

```
int x = 2;
int prod = 1;
for (int n = 5; n > 0; n /= 2) {
  if (n % 2 != 0)
    prod *= x;
  x = x * x;
}
cout << prod << '\n';
```

```
int n = 0;
while (n < 10)
 cout << n << '\n';
```

```
int n = 0;
while (n > 10)
 cout << n << '\n';
```

```
int n = 0;
do {
 cout << n << '\n';
} while (n > 5);
```

```
for (int n = 13; n > 1; n = n % 2 == 0 ? n / 2 : n * 3 + 1)
  cout << n << ' ';
cout << '\n';
```

Switch statement

```
for (int i = 0; i <= 10; ++i)
  switch(i) {
  case 0: cout << "zero "; break;
  case 1: cout << "one "; break;
  case 2: cout << "two "; break;
  case 3: cout << "three "; break;
  case 4: cout << "four "; break;
  case 5: cout << "five "; break;
  case 6: cout << "six "; break;
  case 7: cout << "seven "; break;
  case 8: cout << "eight "; break;
  case 9: cout << "nine "; break;
  default: cout << "\n"; break;
```

computation and overflow

```
double f(int x) {
  return x*x;
}

double g(double x) {
  return x*x;
}

int main() {
  int n = 1000000;
  cout << f(n) << '\t' << g(n) << '\n';
```

Infinity and NaN

```
double a = 0.0;
double b = 1.0;
double c = b / x;
double d = -b / x;
double e = x / x;
double f = sin(x);
double g = tan(x);
double h = sqrt(x);
cout << a << ' ' << b << ' ' << c << ' ' << d << '\n';
cout << e << ' ' <<  f << ' ' << g << ' ' << h << '\n' ;
```

logical operators

```
int a = 2, b = 3;

if (a < 3 && b < 3)
  cout << "A";
else
  cout << "B";

if (a > 3 || b <= 3)
```

```
  cout << "C";
else
  cout << "D";

if (a != 3 && b == 3)
  cout << "E";
else
  cout << "F";
```

Short-circuiting of Logical Operators

```
bool f() {
  cout << "f";
  return true;
}

int main() {
  int a = 2;
  if (a < 5 || f())
     cout << "A";
  if (a < 5 && f())
     cout << "B";
  if (a > 5 || f())
     cout << "C";
  if (a > 5 && f())
     cout << "D";
}
```