Week 1: Introduction to C++

CSCI 2100 Data Structures

Department of Computer Science
Saint Louis University

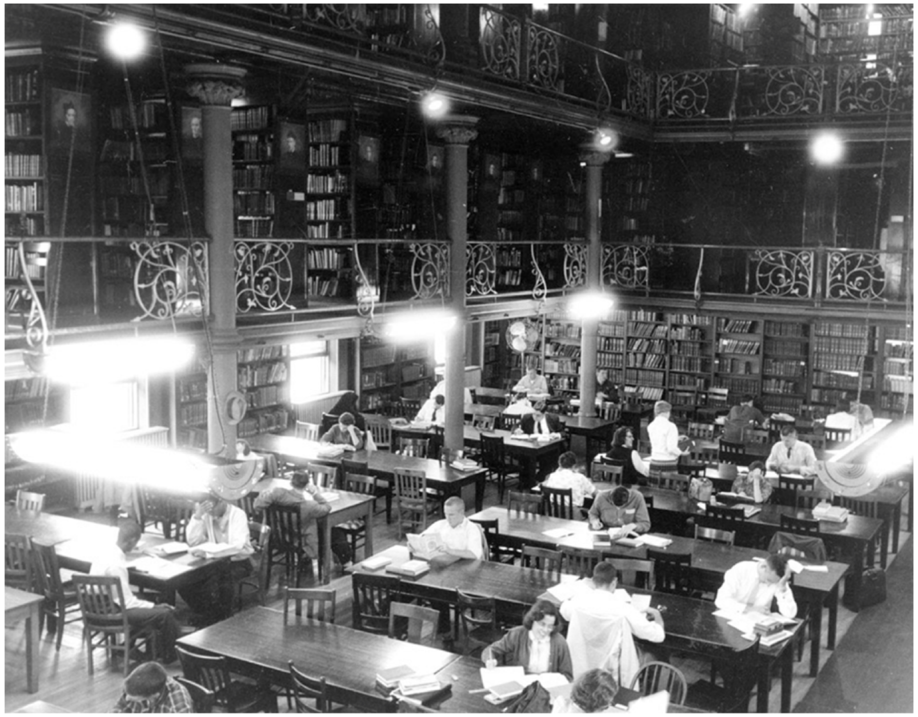
Syllabus

- Announcements
  - Syllabus (Please read in detail!!)
  - Lab on Fridays

Resources for this class

- Data Structures and Algorithms in C++ (Second Edition) Michael T. Goodrich, Roberto Tamassia and David M. Mount John Wiley & Sons, 2011. ISBN-13 978-0-470-38327-8.
- A Transition Guide from Python 2.x to C++ by Dr. Goldwasser and Dr. Letscher.
- www.cplusplus.com
- Tutoring and office hours

Introduction to Data Structures


Students studying in the old DuBourg library reading room, 1958

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

Linus Torvalds *

* The creator, and historically, the principal developer of the Linux kernel.

The Need for Data Structures

A data structure is a way to store and organize data in a program, so that it can be used efficiently.

- Data structures organize data, allowing more efficient programs.
- Applications complexity increases with computation power; demand more calculations.
- Complex computing tasks are unlike our everyday experience.

Efficiency

- A solution is said to be efficient if it solves the problem within its resource limits.
  - Space
  - Time

- The cost of a solution is the amount of resources that the solution consumes.

Abstract Data Types

An abstract data type (ADT) is a type whose behavior is defined by a set of values and a set of operations.

It is about **what** operations are to be performed but not **how** these operations will be implemented.

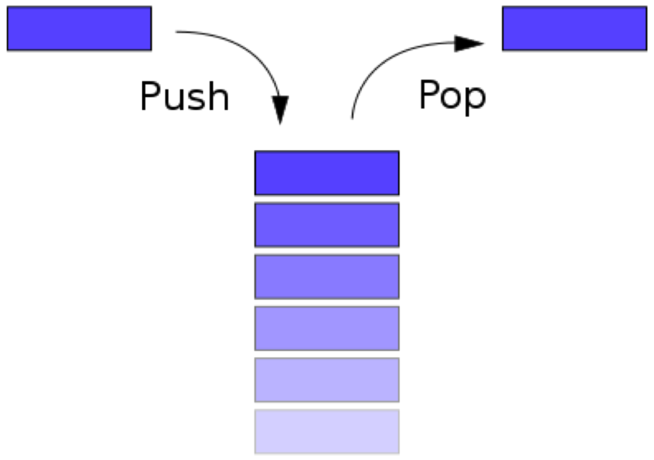
It is called "**abstract**" because it gives an implementation independent view.

Advantages:

- Encapsulation: Hide implementation details.
- Localization of change: Changing ADT does not change the code.

Terminology examples

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context.



- Supporting *push*, *pop*, *size*, and *empty* operations.
- Stacks can be implemented using arrays and linked list.
- Class template using std::stack.

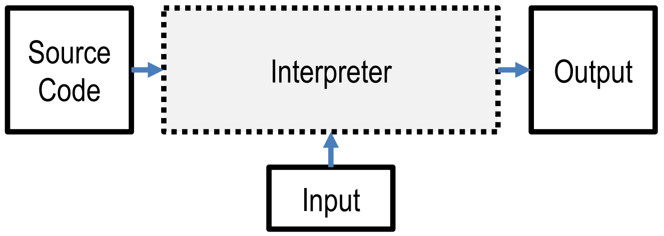
C++

- In 1979, Bjarne Stroustrup, a Danish computer scientist, began work on "C with Classes", the predecessor to C++
- In 1983, "C with Classes" was renamed to "C++" (++ being the increment operator in C), adding new features
- It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.

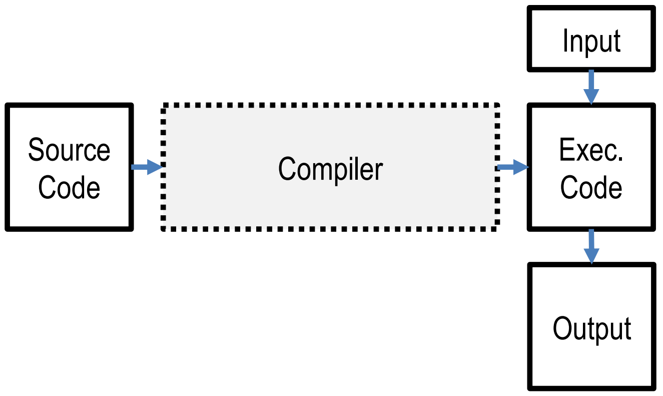
```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
    return 0;
}
```

Python vs C++

- High level
  - Readable
- Dynamically typed
- Industrial strength for rapid development
- Interpreted



- Low level
  - Close to machine code
- Statically typed
- Useful for performance intensive tasks
- Compiled



Why learn C++?

- Faster - Python is about 10 to 100 times slower as compared to C++
- More control over your resources such as memory

Comparison

Python

```
def gcd(u, v):
   # we will use Euclid's algorithm
   # for computing the GCD
   while v != 0:
      r = u % v     # compute remainder
      u = v
      v = r
   return u

if __name__ == '__main__':
   a = int(raw_input('First value: '))
   b = int(raw_input('Second value: '))
   print 'gcd:', gcd(a,b)
```

C++

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

int gcd(int u, int v) {
   /* We will use Euclid's algorithm
      for computing the GCD */
   int r;
   while (v != 0) {
      r = u % v; // compute remainder
      u = v;
      v = r;
   }
   return u;
}

int main( ) {
   int a, b;
   cout << "First value: ";
   cin >> a;
   cout << "Second value: ";
   cin >> b;
   cout << "gcd: " << gcd(a,b) << endl;
   return 0;
}
```

White Space

- Returns, tabs, etc. are ignored in C++

```
int gcd(int u, int v) { int r; while (v != 0) { r = u % v; u = v; v = r; } return u; }
```

- Recall that these were very import in Python
- Here, we use ( ) and { } to mark loops, Booleans, etc.

Compiling

- In Python, you save code as gcd.py and then type "python gcd.py" to run it
- In C++:
  - Save as gcd.cpp
  - Type "`g++ -o gcd gcd.cpp`"
  - Type "`./gcd`"
  - You can also compile as a simple as "`g++ gcd.cpp`", then it saves executable as a.out. Then type "`./a.out`".
  - Filename extensions for C++ are (a) .cc, (b) .cpp, (c) .cxx, (d) .c++, (e) .h, (f) .hh, (g) .hpp, (h) .hxx, and (i) .h++.

Data Types

The precise number of bits devoted to these types is system-dependent, with typical values shown as below.

C++ Type	Description	Literals	Python analog
bool	logical value	true false	bool
short	integer (often 16 bits)		
int	integer (often 32 bits)	39	
long	integer (often 32 or 64 bits)	39L	int
——	integer (arbitrary-precision)		long
float	floating-point (often 32 bits)	3.14f	
double	floating-point (often 64 bits)	3.14	float
char	single character	'a'	
string[a]	character sequence	"Hello"	str

Recap Python2.x data types at https://docs.python.org/2/library/stdtypes.html

Data Types

- Integers can also be ***unsigned*** (non-negative numbers)
  - Signed Int: from – `(`2^{b-1}`)` to + `(`2^{b-1}`−1)`
  - Unsigned Int: from `0` to + `(`2^{b}`−1)`
- C++ also supports two different floating-point types, ***float*** and ***double***, with a ***double*** historically represented using twice as many bits as a ***float***.
- In C++, the ***double*** is most commonly used and akin to what is named float in Python.
- *Finally, we note that Python's **long** type serves a completely different purpose, representing integers with unlimited magnitude. There is no such standard type in C++.*

Character strings

- The ***char*** type provides an efficient representation of a single character of text, while the ***string*** class serves a purpose similar to Python's ***str*** class, representing a sequence of characters (an empty string or a single-character string).

To distinguish between a ***char*** and a one-character ***string***:

string uses double quote "a"	***char*** uses single quote 'a'

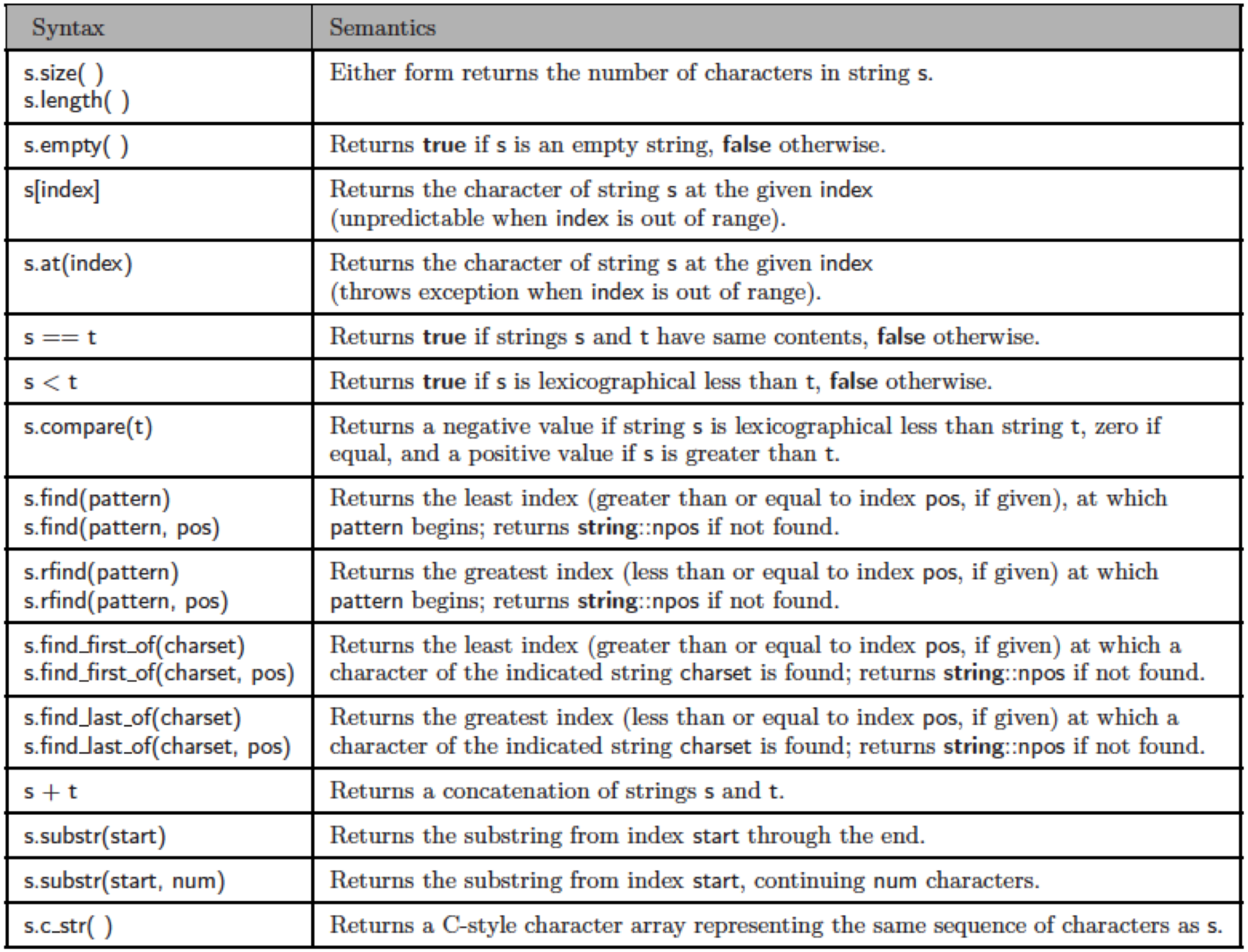
Character strings

- The ***string*** class is not a built-in type; it must be included from among the standard C++ libraries.

```
char a;
a = ‘a’;
a = ‘h’;
```

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

string word;
word = “CS2100”
```

String class: non-mutating behaviors



Syntax	Semantics

s.size( )
s.length( )
	Either form returns the number of characters in string s.

s.empty( )	Returns **true** if s is an empty string, **false** otherwise.
s[index]	Returns the character of string s at the given index (unpredictable when index is out of range).
s.at(index)	Returns the character of string s at the given index (throws exception when index is out of range).
s == t	Returns **true** if strings s and t have same contents, **false** otherwise.
s < t	Returns **true** if s is lexicographical less than t, **false** otherwise.
s.compare(t)	Returns a negative value if string s is lexicographical less than string t, zero if equal, and a positive value if s is greater than t.

s.find(pattern)
s.find(pattern, pos)
	Returns the least index (greater than or equal to index pos, if given), at which pattern begins; returns **string**::npos if not found.

s.rfind(pattern)
s.rfind(pattern, pos)
	Returns the greatest index (less than or equal to index pos, if given) at which pattern begins; returns **string**::npos if not found.

s.find_first_of(charset)
s.find_first_of(charset, pos)
	Returns the least index (greater than or equal to index pos, if given) at which a character of the indicated string charset is found; returns **string**::npos if not found.

s.find_last_of(charset)
s.find_last_of(charset, pos)
	Returns the greatest index (less than or equal to index pos, if given) at which a character of the indicated string charset is found; returns **string**::npos if not found.

s + t	Returns a concatenation of strings s and t.
s.substr(start)	Returns the substring from index start through the end.
s.substr(start, num)	Returns the substring from index start, continuing num characters.
s.c_str( )	Returns a C-style character array representing the same sequence of characters as s.

String class: mutating behaviors

Syntax	Semantics
s[index] = newChar	Mutates string s by changing the character at the given index to the new character (unpredictable when index is out of range).
s.append(t)	Mutates string s by appending the characters of string t.
s += t	Same as s.append(t).
s.insert(index, t)	Inserts copy of string t into string s starting at the given index.
s.insert(index, num, c)	Inserts num copies of character c into string s starting at the given index.
s.erase(start)	Removes all characters from index start to the end.
s.erase(start, num)	Removes num characters, starting at given index.
s.replace(index, num, t)	Replace num characters of current string, starting at given index, with the first num characters of t.

Arrays

- The standard structure for storing a mutable sequence of values in Python is the **list** class.

- C++ also supports a more low-level sequence, known as an **array**, which has its origin in the C programming language.

- What makes an array different from a structure such as a Python list is that the ***size of the array must be fixed*** when the array is constructed and that the contents of the array must have the ***same data type***.

Declarations and Initialization

```
int r;
```

```
int a,b;
```

```
int age(42);
```

```
int age(curYear − birthYear);
```

```
int age(42), zipcode(63103);        // two new variables
```

```
string response;                // guaranteed to be the empty string ""
string greeting("Hello");       // initialized to "Hello"
string rating(3, 'A');          // initialized to "AAA"
```

```
double measurements[300];
```

```
int daysInMonth[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

```
char greeting[ ] = "Hello";
```

Mutability

- Python offers a list class for representing mutable sequences, and a tuple class for representing immutable sequences.
- In C++, all types are assumed to be mutable, but particular instances can be designated as immutable by the programmer.
- Syntactically, the immutability is declared using the keyword **const** in specific contexts.

```
const int age(42);     // immortality
```

Operators

	Python	C++	Description
		Arithmetic Operators	
	−a	−a	(unary) negation
	a + b	a + b	addition
	a − b	a − b	subtraction
	a ∗ b	a ∗ b	multiplication
▷	a ∗∗ b		exponentiation
	a / b	a / b	standard division (depends on type)
▷	a // b		integer division
	a % b	a % b	modulus (remainder)
▷		++a	pre-increment operator
▷		a++	post-increment operator
▷		−−a	pre-decrement operator
▷		a−−	post-decrement operator
		Boolean Operators	
▷	**and**	&&	logical and
▷	**or**	||	logical or
▷	**not**	!	logical negation
▷	a **if** cond **else** b	cond ? a : b	conditional expression

	Python	C++	Description
		Comparison Operators	
	a < b	a < b	less than
	a <= b	a <= b	less than or equal to
	a > b	a > b	greater than
	a >= b	a >= b	greater than or equal to
	a == b	a == b	equal
▷	a < b < c	a < b && b < c	chained comparison
		Bitwise Operators	
	˜a	˜a	bitwise complement
	a & b	a & b	bitwise and
	a | b	a | b	bitwise or
	a ^ b	a ^ b	bitwise XOR
	a << b	a << b	bitwise left shift
	a >> b	a >> b	bitwise right shift

Converting between types

- There are similar settings in which C++ implicitly casts a value to another type.
- Be careful!

```
int a(5);
double b;
b = a;              // sets b to 5.0

int a(4), b(3);
double c;
c = a/b;            // sets c to 1.0
c = double(a)/b     // sets c to 1.33
```

- Converting with strings will be discussed soon.

Control Structures

- C++ has loops, conditionals, functions, and objects.

- Syntax is similar, but just different enough to get into trouble.

- Remembers to use *cplusplus.com* or the *transition guide*.

While Loop

```
while (bool)
{
     body;
}
```

Careful!!

```
int x(20);
while (x<0)
    x = x-5;
    cout << x;
```

$\longrightarrow$

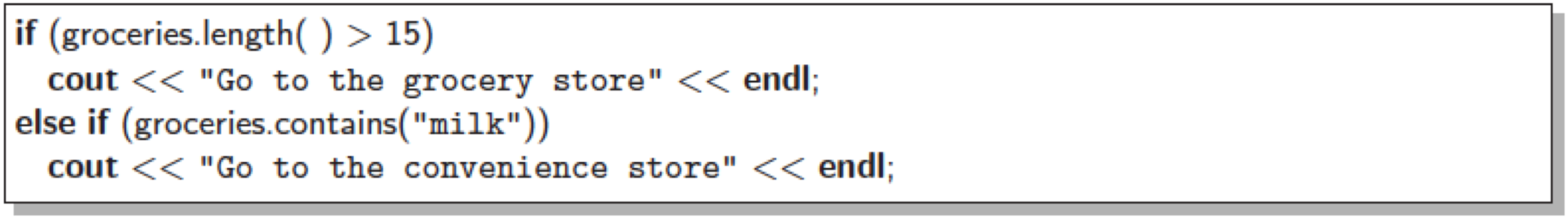
```
int x(20);
while (x<0)
    x = x-5;
cout << x;
```

- Notes:
  - bool is any boolean expression.
  - Don’t need {} if only one command in the loop.

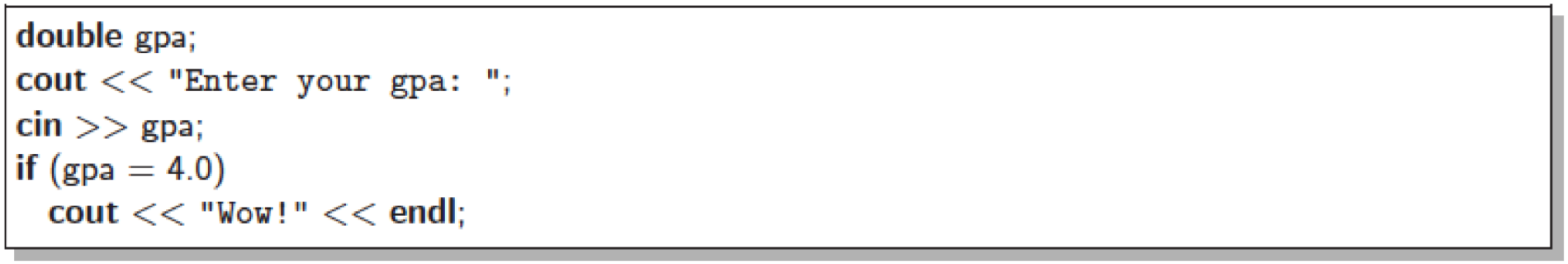
Conditionals

```
if(bool) {
    body1;
}
else {
    body2;
}
```

- Notes: No ***elif***



```
if (groceries.length( ) > 15)
  cout << "Go to the grocery store" << endl;
else if (groceries.contains("milk"))
  cout << "Go to the convenience store" << endl;
```



```
double gpa;
cout << "Enter your gpa:  ";
cin >> gpa;
if (gpa = 4.0)
  cout << "Wow!" << endl;
```

For Loops

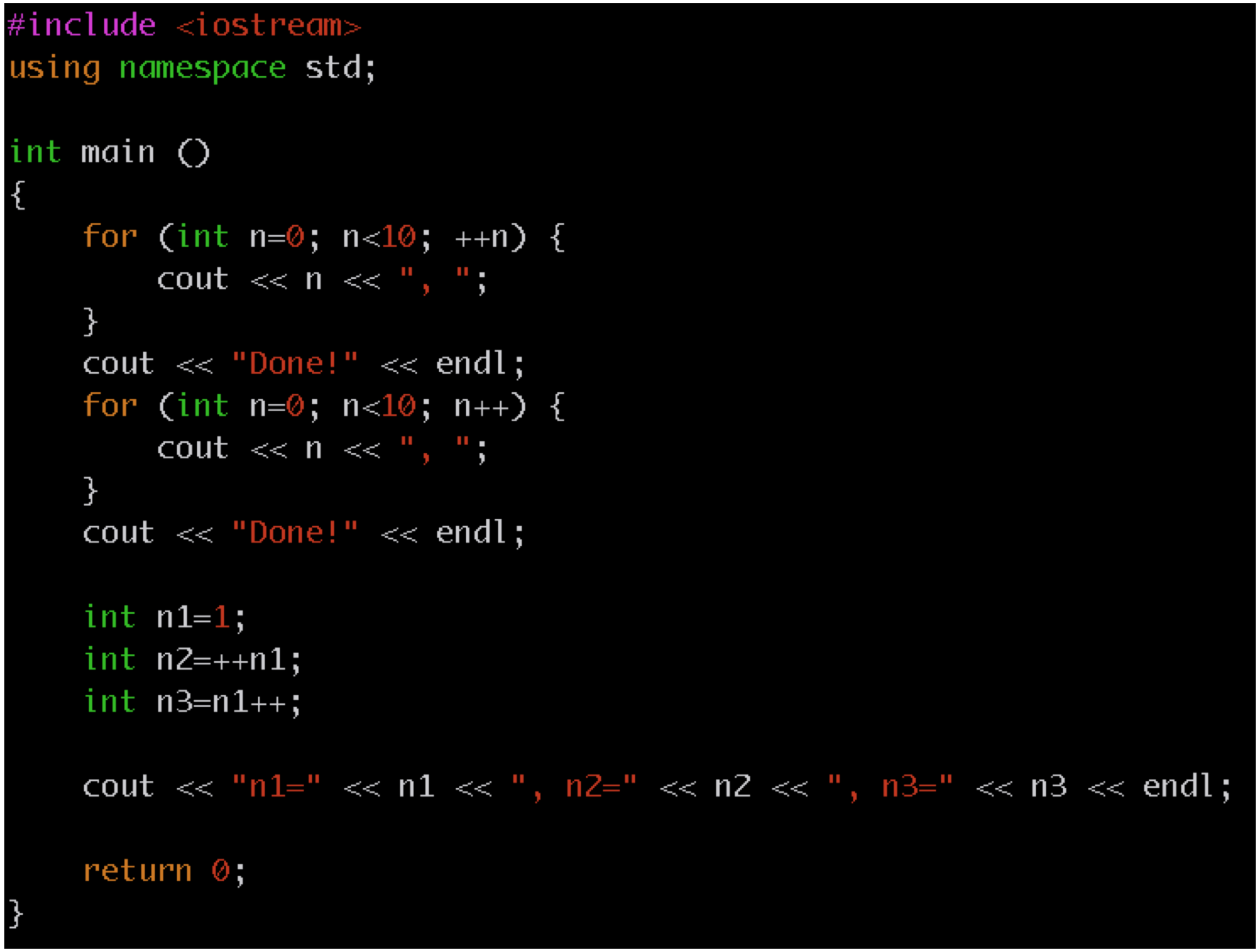
1. *Initialization*
2. *Loop condition*
3. *Update*

```
for (int count = 10; count > 0; count--)
  cout << count << endl;
cout << "Blastoff!" << endl;
```

Note: ***int*** declaration isn't required if the variable was already declared before.

```
int count, total;
for (count = 1, total = 0; count <= 10; count++)
  total += count;
```

Increment / Decrement Operators



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

int main ()
{
    for (int n=0; n<10; ++n) {
        cout << n << ", ";
    }
    cout << "Done!" << endl;
    for (int n=0; n<10; n++) {
        cout << n << ", ";
    }
    cout << "Done!" << endl;

    int n1=1;
    int n2=++n1;
    int n3=n1++;

    cout << "n1=" << n1 << ", n2=" << n2 << ", n3=" << n3 << endl;

    return 0;
}
```

- What’s the results?

Defining a Function

- Every C++ program has at least one function, which is **main()**, and most of the trivial programs can define additional functions.

Defining a Function

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition consists of a function **header** and a function **body**.

Return Type − A function may return a value. The return type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword **`void`**.

Defining a Function

```
// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Calling a Function

```
// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

```
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // calling a function to get max value.
   ret = max(a, b);
   cout << "Max value is : " << ret << endl;

   return 0;
}
```

Function Declaration

A function declaration tells the compiler about a **function name** and **how to call the function**. The actual body of the function can be defined separately.

A function declaration (signature) has the following parts

```
return_type function_name( parameter list );
```

Example:

```
int max(int num1, int num2);
```

Function declaration is **required** when you define a function in one source file and you call that function in another file.

Function Optional Parameters

```
void countdown( ) {
  for (int count = 10; count > 0; count--)
    cout << count << endl;
}
```

```
void countdown(int start=10, int end=1) {
  for (int count = start; count >= end; count--)
    cout << count << endl;
}
```

countdown(5,2): form 5 to 2

countdown(8): from 8 to 1

countdown( ): from 10 to 1

Input and Output

- Necessary libraries
  - Technically, streams are not automatically available in C++. Rather, they are included from one of the standard libraries.

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

$\longrightarrow$ Optional

Class	Purpose	Library
istream	Parent class for all input streams	<iostream>
ostream	Parent class for all output streams	<iostream>
iostream	Parent class for streams that can process input and output	<iostream>
ifstream	Input file stream	<fstream>
ofstream	Output file stream	<fstream>
fstream	Input/output file stream	<fstream>
istringstream	String stream for input	<sstream>
ostringstream	String stream for output	<sstream>
stringstream	String stream for input and output	<sstream>

Input and Output

- The **cout** identifier represents a special output stream used to display information to the user console.
- The **<<** symbol is an operator for inserting data into that output stream.
- We use the **cin** object to read input from the user console.
- The **>>** operator is used to extract information from the stream into a variable.

Output

Python

```
print "Hello"
print                       # blank line
print "Hello,", first
print first, last           # automatic space
print total
print str(total) + "."      # no space
print "Wait...",            # space; no newline
print "Done"
```

C++

```
cout << "Hello" << endl;
cout << endl;                       // blank line
cout << "Hello, " << first << endl;
cout << first << " " << last << endl;
cout << total << endl;
cout << total << "." << endl;
cout << "Wait... ";                 // no newline
cout << "Done" << endl;
```

Formatting

```
cout << "pi is " << fixed << setprecision(3) << pi << endl;
```

This command would result in the output pi is 3.142.

```
cout << left << setw(10) << item << " "<< right << setw(5) << quantity << endl;
```

we get a result of

```
pencil       50
pen         100
```

Console Input

```
person = raw_input('What is your name?)
```

```
int number;
cout << "Enter a number from 1 to 10: ";   // prompt without newline
cin >> number;                             // read an integer from the user
```

- Notes:
  - inputs are separated by any white space
  - type of input must match type of variable

Console Input

- Problem:

```
string person;
cout << "What is your name? ";        // prompt the user (without a newline)
cin >> person;                        // input the response
```

```
What is your name? John Smith
```

  - After executing the C++ code, the variable person will be assigned the string "John", while the subsequent characters (" Smith\n") remain on the stream.

- **getline()** : function to save the string up to the next new line

```
string person;
cout << "What is your name? ";        // prompts the user (without a newline)
getline(cin, person);                 // stores result directly in variable 'person'
```

Tricky Input Example

```
int age;
string food;
cout << "How old are you? ";
cin >> age;
cout << "What would you like to eat? ";
getline(cin, food);
```

A typical user session might proceed as follows.

```
How old are you?  42
What would you like to eat? pepperoni pizza
```

The problem is that after executing the above code, the variable food will be set to the empty string `""`.

How to solve this problem? Add a line "`cin.ignore();`" after "`cin >> age;`"

File Streams

If the name of an existing file is known:

```
ifstream mydata("scores.txt");
```

If the name of an existing file is unknown:

```
ifstream mydata;
string filename;
cout << "What file? ";
cin >> filename;
mydata.open(filename.c_str( ));        // parameter to open must be a C−style string
```

Note: Writing a file default: overwrite.
To append:

```
ofstream datastream("scores.txt", ios::app);
```

String Streams

- Casting between numbers and strings

```
int age(42);
string displayedAge;
stringstream ss;
ss << age;            // insert the integer representation into the stream
ss >> displayedAge;   // extract the resulting string from the stream
```

Error Checking with Input Streams

```
number = 0;
while (number < 1 || number > 10) {
  cout << "Enter a number from 1 to 10: ";
  cin >> number;
  if (cin.fail( )) {
    cout << "That is not a valid integer." << endl;
    cin.clear( );                                              // clear the failed state
    cin.ignore(std::numeric_limits<int>::max( ), '\n');       // remove errant characters from line
  } else if (cin.eof( )) {
    cout << "Reached the end of the input stream" << endl;
    cout << "We will choose for you." << endl;
    number = 7;
  } else if (cin.bad( )) {
    cout << "The input stream had fatal failure" << endl;
    cout << "We will choose for you." << endl;
    number = 7;
  } else if (number < 1 || number > 10) {
    cout << "Your number must be from 1 to 10" << endl;
  }
}
```