# Week 1

# Chapter 0: Introduction

Outline:

- History of C++
- High level differences
- Reasons to use C++

# History

- In 1972 Dennis Ritchie designed and implemented C to write unix.
- In 1979 Bjarne Stroustrup started work on C++ to have “C with classes”
- Around 1994 James Gosling developed Java

# Big Differences

- C++ programs are turned into Completely compiled binary code run on a real machine, but Java programs are turned into partially compiled bytecode interpreted by a virtual machine.
- Java is more for safety but C++ is more for efficiency.
- Java supports native multithreading but C++ does not.
- Java provides comprehensive application support with packages but C++ gives container support with the standard template library (STL).

# Why C++

- C++ programming is a skill in high demand.
- C++ provides templates for generic programming.
- C++ supports operator overloading to operate on objects.
- C++ allows conditional compilation.
- C++ distinguishes between accessors and mutators.
- C++ aims for time and space efficiency.

# Chapter 1: Basic types

Outline:

- First program
- Primitive Types
- Minor Syntactic Differences

# First Program

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello world" << endl;
    return 0;
}
```

# Compile and Execute

```
$cd
$mkdir cs1280
$cd cs1280
$mkdir lect
$cd lect
$mkdir 1
$cd 1
$ vi 1.cpp
$ ls
1.cpp
```

```
$ g++ 1.cpp
$ ls
1.cpp a.out
$ a.out
Hello world
$ echo $? # check return value of main()
0
$ rm a.out
rm: remove a.out (yes/no)? y
$ ls
1.cpp
```

# #include <iostream>

- The system header file iostream (<iostream>) is inserted into the source as directed by the preprocessing directive (#include).
- iostream contains declarations and definitions for the standard I/O classes and objects.

Explore:

To see these declarations and definitions, enter

```
g++ -E 1.cpp
```

# using namespace std

- The using namespace directive is like a Java import statement
- std is the name of a namespace
- using namespace std; allows the abbreviation of

  std::cout as cout

  and

  std::endl as endl

# int main()

- A C++ program starts execution at the main() function
- The main() function should return an int value
- The return value of main() is stored in the shell variable $?
- main() may have up to three parameters

- A java executable starts execution at the static void main() method that returns nothing.
- To return a value to the system a java executable calls System.exit().

# cout, endl

- The object cout is of class ostream that represents the standard output stream, one of the three standard I/O streams of unix.
- The insertion operator << inserts its right operand to its left operand.
- The manipulator endl inserts a newline character and flushes the buffer.

1. The expression cout << operand evaluates to cout and the operator << is left-associative.
2. Thus the right operands of << can be cascaded and

   ```
   cout << one << two << three;
   ```

   is the same as

   ```
   ((cout << one) << two) << three;
   ```

# Primitive Types

- The basic data types of C++ are integer, floating-point, character, and boolean.
- The actual size of a basic C++ type may be platform dependent.
- The size of a type *Type* is obtained by the compile-time operator sizeof(*Type*).

  e.g. sizeof(int)
- sizeof can also take variable names or expression: sizeof (*expression)*

  e.g. sizeof ('\n')

# The Size of Some Primitive Types

```
$ cat sizeof.cpp
#include <iostream>
using namespace std;
int main() {
      cout << sizeof(short) << ’ ’
      << sizeof(int) << ’ ’
      << sizeof(long) << ’ ’
      << sizeof(long long) << ’ ’ << endl;
      cout << sizeof(float) << ’ ’
      << sizeof(double) << ’ ’
      << sizeof(long double) << ’ ’ << endl;
      cout << sizeof(char) << sizeof(bool) << endl;
      cout << sizeof('\n‘) << ’ ’ << sizeof("\n“)  << endl;
      int a,b;
      cout << sizeof (a+b) << endl; return 0;
}
```

```
$ a.out
2 4 4 8
4 8 16
1 1
1 2
4
```

# Integers

- The integer type is int.
- The size can be modified with short, long, and long long.
- An integer can be signed or unsigned.
- The range of an n-bit 2’s complement signed integer type is

  $-2^{n-1} .. 2^{n-1} – 1$
- The range of an n-bit unsigned integer type is

  $0 .. 2^{n} – 1$

# Sizes and Ranges of Signed Integers

```
#include <iostream>
#include <climits>             // min and max of signed integer types
using namespace std;
int main() {
      cout << sizeof(short) <<’\t’<< SHRT_MIN << ".." << SHRT_MAX
          << endl;
      cout << sizeof(int) <<’\t’<< INT_MIN << ".." << INT_MAX << endl;
      cout << sizeof(long) <<’\t’<< LONG_MIN << ".." << LONG_MAX <<
          endl;
      cout << sizeof(long long) <<’\t’<< LLONG_MIN << ".." <<
          LLONG_MAX << endl;
      return 0;
}
```

```
$ g++ signed.cpp
$ a.out
2         -32768..32767
4         -2147483648..2147483647
4         -2147483648..2147483647
8         -9223372036854775808..9223372036854775807
```

# Overflow and Underflow (1)

- Overflow: value too large
- Underflow: value too small

```
#include <iostream>
using namespace std;
int main(){
  short i = 32767;                    // largest value
  cout << "i: " << i << endl;
  i = i+1;
  cout << "i+1: " << i << endl; // Overflow, become smallest value
  i = i-1;
  cout << "i-1: " << i << endl; // underflow, become largest value
  return 0;
}
```

```
$ g++ overflow.cpp
$ a.out
i: 32767
i+1: -32768
i-1: 32767
```

# Overflow and Underflow (2)

```
#include <iostream>
using namespace std;
int main(){
 float test;
  test = 2.0e38 * 1000;
  cout << test << endl;
  test = 2.0e-38 / 2.0e38;
  cout << test << endl;  return 0;
}
```

```
$ g++ overflow1.cpp
$ a.out
Inf
0
```

# Sizes and Ranges of Unsigned Integers

```
#include <iostream>
#include <climits>                                    // max and min of integer types
using namespace std;
int main() {
     unsigned short us = 2*SHRT_MAX+1; short ss = us;
     unsigned int ui = 2*INT_MAX+1; int si = ui;
     unsigned long ul = 2*LONG_MAX+1; long sl = ul;
     unsigned long long ull = 2*LLONG_MAX+1; long long sll = ull;
     cout << sizeof(unsigned short) <<’\t’<< ss << ’\t’ << us << endl;
     cout << sizeof(unsigned int) <<’\t’<< si << ’\t’ << ui << endl;
     cout << sizeof(unsigned long) <<’\t’<< sl << ’\t’ << ul << endl;
     cout << sizeof(unsigned long long) <<’\t’<< sll << ’\t’ << ull << endl;
     return 0;
}
```

# Output: The Sizes and Ranges of Unsigned Integers

```
$ g++ unsigned.cpp
$ a.out
2 -1 65535
4 -1 4294967295
4 -1 4294967295
8 -1 18446744073709551615
```

# Floating-Points

- The floating-point format allows the separatioin of magnitude and precision.
- The floating-points are float, double, and long double.

# Sizes and Ranges of Floating-Points

```
#include <iostream>
#include <cfloat>              // min and max of floating-point types
using namespace std;
int main() {
     cout << sizeof(float) << ’\t’ << FLT_MIN <<" .. "<< FLT_MAX <<
        endl;
     cout << sizeof(double) << ’\t’ << DBL_MIN <<" .. "<< DBL_MAX
        << endl;
     cout << sizeof(long double) << ’\t’ << LDBL_MIN <<" .. "<<
        LDBL_MAX << endl;
     return 0;
}
```

```
$ g++ real.cpp
$ a.out
4          1.17549e-38 .. 3.40282e+38
8          2.22507e-308 .. 1.79769e+308
16         3.3621e-4932 .. 1.18973e+4932
```

# Characters

- C++ supports the 128 7-bit ASCII characters.
- The backslash \ escape character is used for non-drawable ASCII characters and some special (meta) characters:

  \n, \t, \v, \b, \r, \f, \a, \\, \?, \’, \", \0, \ooo, \xhhh.

# Program: Characters –(1)

```
#include <iostream>
using namespace std;
const char T(’\t’);                    // or T = ’\t’
int main() {
      cout << sizeof(char) << T
      << ’\137’ << T << ’\x5f’ << T << endl;
      return 0;
}
```

```
$ g++ char.cpp
$ a.out
1           _           _
```

# Program: Characters –(2)

```
$ cat ascii.cpp
#include <iostream>
using namespace std;
int main() {
  cout
    << '\74'
    << '\074'
    << '\x3c'
    << (char) 60
    << '<'
    << endl;
  return 0;
}
```

```
$ g++ ascii.cpp
$ a.out
<<<<<
```

# Overflow revisited

- What will happen when the following program is run?

```
#include <iostream>
using namespace std;
int main(){
  char i;
  for(i=0; i<256; i++)
    cout << (int)i << endl ;
  return 0;
}
```

It will loop forever!

Overflow and underflow are not regarded as errors that can crash the program

# Boolean Values

- The boolean type is bool.
- C++ represents true with 1 and false with 0.
- C++ takes non-zero values as true and zero as false.

# Program: Boolean Values

```
#include <iostream>
int main() {
        std::cout << sizeof(bool) << '\n';
        bool f = false;
        std::cout
        << f << '\t' << true << '\t'
        << (0==1) << '\t' << (0!=1) << '\t'
        << (bool) -1 << '\t' << (bool) 2 << '\t'
        << (bool)(1.0) << std:: endl;
        return 0;
}
```

```
$ g++ bool.cpp
$ a.out
1
0       1       0       1       1       1       1
```

# Minor Syntactic Differences

- C++ takes both boolean and numeric expressions (0 false, non-0 true) as conditional expression
- C++ does less than java in ensuring variables are initialized and a value is returned when one is needed.

# Chapter 2: Functions, Arrays, Strings, Parameter passing

Outline:

- Functions
- Arrays and Strings
- Parameter passing

# Functions and Non-class Functions

- A function definition consists of a return type, a function name, a parameter list, and a body enclosed by braces.
  - E.g.: int add2( int a) { return a+2; }
- A function is invoked by its name with the necessary arguments.
- Non-class functions are “stand-alone” functions that do not belong to any class.
- A non-class function has a global scope and thus it is also called a global function.

# Function Signature and Overloading

- The function name, number of parameters and their types, are collectively known as the signature of the function.
- A function is identified by its signature.
  - We can't have 2 functions with the same signature but different return type.
  - We can have 2 functions of different signatures yet with the same function name.
- Allowing more than one function to have the same name is known as function overloading.
- Overloaded functions are distinguished by their parameters.

Can we have 2 functions of different signatures yet with the same set of statements in the function body?

# Prototype - Function Declaration

- To compile, a function must either be declared or defined before its invocation.
- To declare a function is to specify its prototype.
- The prototype of a function is like the definition of a function.
  - But in a prototype, parameter names can be omitted and the function body is replaced by a semi-colon.

# Prototype:
# parameter names are optional

Example:

```
triangle(double, double, double);  // ok, but
// Its is clearer when meaningful names are used
triangle(double angleA, double sideB, double angleC);
```

# Function Declaration Example(1)

```
$ cat prototype.cpp
#include <iostream>
using namespace std;
int add2(int);
int main() { cout << add2(1) << endl; return 0; }
int add2(int a) { return a+2; }
```

```
$ g++ prototype.cpp
$ a.out
3
```

# Function Declaration Example(2)

```
$ cat prototype2.cpp
#include <iostream>
using namespace std;
int add2(int a) { return a+2; }
int main() { cout << add2(1) << endl; return 0; }
```

# Default Parameter Values

- When a function is invoked, some actual parameter values may be omitted if the corresponding formal parameters in the function declaration or definition are given default values.
- Since the syntax does not allow a comma terminated null, if a formal parameter is given a default value, all subsequent formal parameters should also be given default values.

# Default Parameter Values Example

```
$ cat defpar.cpp
#include <iostream>
using namespace std;
int f(int a=-1, int b=-2, int c=-3) { return a+b+c; }
int main() {
      cout << f() <<’ ’<< f(1) <<’ ’<< f(1,2) <<’ ’<< f(1,2,3)
          << endl;
      return 0;
}
```

```
$ g++ defpar.cpp
$ a.out
-6 -4 0 6
```

# Parameter Passing: Call by Value

- In call by value the formal parameter is initialized by the actual parameter.
- The formal parameter and the actual parameter are two separate variables.

# Parameter Passing: Call by Reference

- In call by reference the formal parameter is the actual parameter.
- The formal parameter and the actual parameter are the same variable.
- This is needed if the actual parameter should be changed after the function has returned.

# Parameter Passing:
# Call by Constant Reference

- In call by constant reference the formal parameter is the actual parameter and is immutable.
- The formal parameter and the actual parameter are the same variable.
- This is needed if the actual parameter is large but should not be changed.

# Parameter Passing Example (1)

```
$ cat par.cpp
#include<iostream>
using namespace std;
void incV(int x) {
cout << x; x++; cout << x << endl;
}
void incR(int & x) {
cout << x; x++; cout << x << endl;
}
```

# Parameter Passing Example (2)

```
int main() {
int x = 123;
incV(123);
incV(x);
cout << x << endl;
// incR(123); The actual parameter must be a variable.
incR(x);        // The way in calling is no different from
                // that of call by value
cout << x << endl;
}
```

```
$ g++ par.cpp
$ a.out
123124
123124
123
123124
124
```

# Separate Compilation

- The functions in a single executable can be created in different files, compiled, and tested separately.
- To compile a program source file to produce an object code, use the “-c” option of the compiler g++.
- The file extension for the object code file is “.o”.
- The object code files must be linked to produce the executable (called a.out by default)

# Separate Compilation Example(1)

```
$ cat gcd.h
int gcd(int, int);
$ cat gcd.cpp
int gcd( int m, int n ) {
    if( m < 0 ) m = -m;
    if( n < 0 ) n = -n;
    int r;
    while( n > 0 ) {r = m%n; m = n; n = r; }
    return m;
}
```

# Separate Compilation Example(2)

```
$ cat gcdtest.cpp
#include <iostream>
#include "gcd.h"
using namespace std;
int main() {
    cout << gcd( 30, 0 ) << endl;
    cout << gcd( 0, -105 ) << endl;
    cout << gcd( 30, -105 ) << endl;
}
```

# Separate Compilation Example(3)

```
$ g++ -c gcd.cpp            # create object gcd.o
$ g++ -c gcdtest.cpp        # create object gcdtest.o
$ g++ gcd.o gcdtest.o       # create linked a.out
$ a.out
30
105
15
```

# C-style Arrays

```
$ cat array.cpp
#include <iostream>
using namespace std;
int days[] = {                          // not int[] days !
31, 28, 31, 30, 31, 30,
31, 31, 30, 31, 30, 31
};
int main() {
    cout << days[1] << endl;
    cout << sizeof(days)/sizeof( days[0]) << endl;
}
```

```
$ g++ array.cpp
$ a.out
28
12
```

# Restriction on the use of array

- All elements of a global array are initialized to zero
  - Local arrays have no default initialization values
  - If an array is partially initialized, then the uninitialized elements will be set to zero
- An array cannot be the lhs of an assignment
- An array cannot be extended
- A function cannot return an array
- C++ supports C-style arrays but provides a vector class for better array support.
- The C++ vector class behaves like java ArrayList class.

# Vectors

```
$ cat vector.cpp
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int main() {
      vector<float> mc;
      cout << mc.size() << “\ncapacity: " << mc.capacity()
             << “\nmax_size: " << mc.max_size()<< endl;
      mc.push_back(M_PI);
      cout << " capacity: " << mc.capacity() << endl;
      mc.push_back(M_E);
      cout << " capacity: " << mc.capacity() << endl;
      mc.push_back(M_SQRT2);
      cout << " capacity: " << mc.capacity() << endl;
      mc[3000] = 3000.0;                // Allowed to access?
     cout << mc[3000] << endl;
     cout << mc.size() << endl;        // =3. Surprising?
}
```

```
$ g++ vector.cpp
$ a.out
size: 0
capacity: 0
max_size:
     1073741823
 capacity: 1
 capacity: 2
 capacity: 4
3.14159
2.71828
1.41421
3000
3
```

# Vector: at(), reserve(), resize()

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
cout << "Element " << i << " is " << v[i] << endl; }
```

- Instead of using [], use at():

```
for( int i = 0; i < 10; i++ ) {
cout << "Element " << i << " is " << v.at(i) << endl; }
```

- reserve(): sets the minimum capacity of the vector
- resize(): change the size of the vector

# C-Style Strings

- A C-style string (C-string) is simply an array of characters terminated by the null character (‘\0’)
- A C-style string can be accessed with the declaration

  char * s;                // s refers to a character string
- The command line arguments are organized as an array of C-style strings. So they are accessed with the declaration

  char * av[];         // av is an array of C-style strings

# C-string <--> Number Conversion

- atof: converts a string to a double (not float!)
- atoi: converts a string to an integer
- atol: converts a string to a long

# C-Style Strings: Command Line Arguments

```
$ cat cstring.cpp
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;
int main(int ac, char *av[]) {
    cout << av[0] << endl;
    for(int i=1; i<ac; i++) {
        int n = atoi(av[i]);
        cout << av[i] <<’\t’<< strlen(av[i])
        <<’\t’ << n <<’\t’<< sizeof(n) << endl;
    }
}
```

```
$ g++ cstring.cpp
$ a.out 0 12 345 67890
a.out
0       1       0       4
12      2       12      4
345     3       345     4
67890   5       67890   4
```

# <cstring> (1)

Provides a lot of useful string manipulation functions

- strcpy (char *string1, const char  *string2) :

  Copy string2 into string1

  string1 must have enough space!
- strcat(char *s1, const char *s2):

  Concatenate string2 to the end of string1

  string1 must have enough space!
- strlen(const char *string) :

  Get the length of a string

# <cstring> (2)

- strcmp(const char *string1, const char *string2)

  Return 0 if string1 equals string2, otherwise non-zero

- strdup(const char *string1)

  returns a pointer to a new string that is a duplicate of the string pointed to by string1

- strtok(char *string1, const char *string2):

  Breaks the string pointed to by string1 into a sequence of tokens, each of which is delimited by a character in the string pointed to by string2

  string1 is corrupted by strtok!!!

# Example 1 – strcpy, strcmp, strlen

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
   char first[100];
   strcpy(first, "CS1280");
   cout << first << " length = " << strlen(first) << endl;
   strcat(first, " and me");
   cout << first << " length = " << strlen(first) << endl;
   if ( !strcmp(first, "CS1280 and me")) { // Why use !?
     cout << "equal" << endl;
   }
}
```

```
$ g++ string1.cpp
$ a.out
CS1280 length = 6
CS1280 and me length = 13
equal
```

# Example 2 - strtok

```
#include <cstring>
#include <iostream>
using namespace std;
int main( int args, char **argv) {
  char *delimiters = " ", *t;
  char s1[] = "I am learning how to use strtok";
  t = strtok(s1, delimiters); // first call to strtok
  int numTokens = 1;
  cout << "Token " << numTokens << ": " << t << endl;
  while ( (t = strtok( NULL, delimiters)) != NULL ) {
    numTokens++;
    cout << "Token " << numTokens << ": " << t << endl;
  }
  cout << endl;
}
```

```
$ g++ strtok.cpp
$ a.out
Token 1: I
Token 2: am
Token 3: learning
Token 4: how
Token 5: to
Token 6: use
Token 7: strtok
```

# Example 3 – strtok1

```
#include <cstring>
#include <iostream>
using namespace std;
int main( int args, char **argv) {
  char *delimiters = " ", *t;
  char s1[100];
  gets(s1);                          // Read in one line
  t = strtok(s1, delimiters);  // first call to strtok
  int numTokens = 1;
  cout << "Token " << numTokens << ": " << t << endl;
  while ( (t = strtok( NULL, delimiters)) != NULL ) {
    numTokens++;
    cout << "Token " << numTokens << ": " << t << endl;
  }
}
```

```
$ g++ strtok1.cpp
$ a.out
I like c++
Token 1: I
Token 2: like
Token 3: c++
```

# Strings

- Functions in <cstring> do not check for index out of range error. As a result, data can be corrupted and programs may crash as a result.
- To solve this problem, C++ provides a string class for better string support.
- It is better to use string unless C-style strings are really required.

# C++ Strings: An Example

```
$ cat string.cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
     string s = "Hello world";
     s += ’.’;
     for(int i=0; i<s.length(); i++)
          cout << s.at(i) << ’_’;
     cout << endl;
     cout << s.compare("Hello world!") << endl;
     cout << s.compare("Hello world.") << endl;
     cout << s.compare("Hello world?") << endl;
}
```

```
$ g++ string.cpp
$ a.out
H_e_l_l_o_ _w_o_r_l_d_._
13
0
-17
```

# Some functions in <string>

- append: append characters and strings
- at: returns the character at a specific location
- c_str: returns a C-string from the string
- compare: compares two strings
- empty: true if the string has no characters
- getline: read data from an I/O stream into a string
- insert: insert characters into a string
- length: returns the length of the string
- substr: returns a certain substring