9. Classes I

13. Standard C++ Classes
14. Custom Objects

String Objects (1)

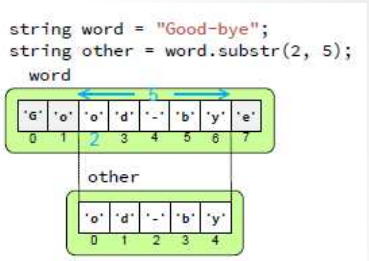
```
#include <string>
using namespace std;
using std::string;

string name = "joe";
std::cout << name << '\n';
name = "jane";
std::cout << name << '\n';

string name1 = "joe", name2;
name2 = name1;
std::cout << name1 << " " << name2 << '\n';
```

String Objects (2)

- operator[]
  - provides access to the value stored at a given index within the string
- operator=
  - assigns one string to another
- operator+=
  - appends a string or single character to the end of a string object
- at
  - provides bounds-checking access to the character stored at a given index
- length
  - returns the number of characters that make up the string
- size
  - returns the number of characters that make up the string (same as length)
- find
  - locates the index of a substring within a string object
- substr
  - returns a new string object made of a substring of an existing string object
- empty
  - returns true if the string contains no characters; returns false if the string contains one or more characters
- clear
  - removes all the characters from a string



String Objects (3)

```
std::string word;
word = "good";
word.operator=("good");
word += "-bye";
word.operator+=("-bye");




const char *c_str() const;
// returns a pointer to an array that contains a null-terminated
sequence of characters (C-string) representing the current value of
the string object
```

Input/Output Streams (1)

```
std::cin >> x;
std::cout << x;

cin.operator>>(x);
cout.operator<<(x);

cout.operator<<(x).operator<<('\n');
```


Input/Output Streams (2)

```
#include <iostream>
#include <limits>
int main() {
   int x;
   std::cout << "Please enter an integer: ";
   while (!(std::cin >> x)) {
      std::cout << "Bad entry, please try again: ";

      std::cin.clear(); // Clear the error state of the stream
      // Empty the keyboard buffer
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
         '\n');
   }
   std::cout << "You entered " << x << '\n';
}
```

Input/Output Streams (3)

```
#include <iostream>
#include <string>
int main() {
   std::string line;
   std::cout << "Please enter a line of text: ";
   std::cin >> line;
   std::cout << "You entered: \"" << line << "\"" << '\n';
}
-----------------------------------------------------------------
#include <iostream>
#include <string>
int main() {
   std::string line;
   std::cout << "Please enter a line of text: ";
   getline(std::cin, line);
   std::cout << "You entered: \"" << line << "\"" << '\n';
}
```


File Streams (1)

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
void print_vector(const std::vector<int>& vec) {
   std::cout << "{";
   int len = vec.size();
   if (len > 0) {
      for (int i = 0; i < len - 1; i++)
         std::cout << vec[i] << ",";
      std::cout << vec[len - 1];
   }
   std::cout << "}\n";
}
```

File Streams (2)

```
void save_vector(const std::string& filename, const std::vector<int>& vec)
{
   std::ofstream out(filename);
   if (out.good()) {
      int n = vec.size();
      for (int i = 0; i < n; i++)
         out << vec[i] << " "; // Space delimited
      out << '\n';
      out.close();
   }
   else std::cout << "Unable to save the file\n";
}
void load_vector(const std::string& filename, std::vector<int>& vec) {
   std::ifstream in(filename);
   if (in.good()) {
      vec.clear();
      int value;
      while (in >> value) vec.push_back(value);
      in.close();
   }
   else std::cout << "Unable to load in the file\n";
}
```


File Streams (3)

```
int main() {
   std::vector<int> list;
   bool done = false;
   char command;

   while (!done) {
      std::cout << "I)nsert <item> P)rint "
         << "S)ave <filename> L)oad <filename> "
         << "E)rase Q)uit: ";

      std::cin >> command;
      int value;
      std::string filename;
      switch (command) {
         case 'I':
         case 'i':
            std::cin >> value;
            list.push_back(value);
            break;
```

File Streams (4)

```
         case 'P':
         case 'p':
            print_vector(list);              break;
         case 'S':
         case 's':
            std::cin >> filename;            save_vector(filename, list);
            break;
         case 'L':
         case 'l':
            std::cin >> filename;            load_vector(filename, list);
            break;
         case 'E':
         case 'e':
            list.clear();                    break;
         case 'Q':
         case 'q':
            done = true;                     break;
      }
   }
}
```


File Streams (5)

```
std::endl      // Inserts a newline character into the output
               // sequence os and flushes it
               // Without std::flush, the output would be the same,
               // but may not appear in real time.

With std::endl (console): 16029
With '\n' (console): 13943
With std::endl (file): 2454
With '\n' (file): 503
```

File Streams (6)

```
#include <iostream>
#include <fstream>
int main() {
    std::ofstream ofs;
    ofs.open("123.bin", std::ios::binary);
    if(ofs.good()) {
        int x = 65;
        ofs.write((const char*)&x, sizeof(int));
        ofs.close();
    }

    std::ifstream ifs;
    ifs.open("123.bin", std::ios::binary);
    if(ifs.good()) {
        int x;
        ifs.read((char *)&x, sizeof(int));
        std::cout << x << std::endl;
        ifs.close();
    }
}
```


File Streams (7)

- Member functions
  - good, eof, fail, bad, operator !
  - tellg, seekg

- Mode:
  - in, out, binary, ate (at end: the output position starts at the end of the fine), app (append), trunc (truncate: any contents that existed in the file before it is open are discarded)

Complex Numbers (1)

```
std::complex<float> fc;
std::complex<double> dc;
std::complex<long double> ldc;

// operator= : assigns the contents
// real : accesses the real part of the complex number
// imag : accesses the imaginary part of the complex number
// operator+=
// operator-=
// operator*=
// operator/=

// non-member functions
// abs, arg, norm, conj, …

using namespace std::complex_literals;
c1 = 1.5 + 2.5i;                              // C++14
```


Complex Numbers (2)

```
#include <iostream>
#include <complex>
int main() {
   std::complex<double> c1(2.0, 3.0), c2(2.0, -3.0);
   double real1 = c1.real(),
      imag1 = c1.imag(),
      real2 = c2.real(),
      imag2 = c2.imag();
   std::cout << c1 << " * " << c2 << " = "
      << real1*real2 - imag1*imag2 << " , "
      << imag1*real2 + real1*imag2
      << '\n';
   std::cout << c1 << " * " << c2 << " = " << c1*c2 << '\n';
}

// http://www.cplusplus.com/reference/complex/
```

String stream

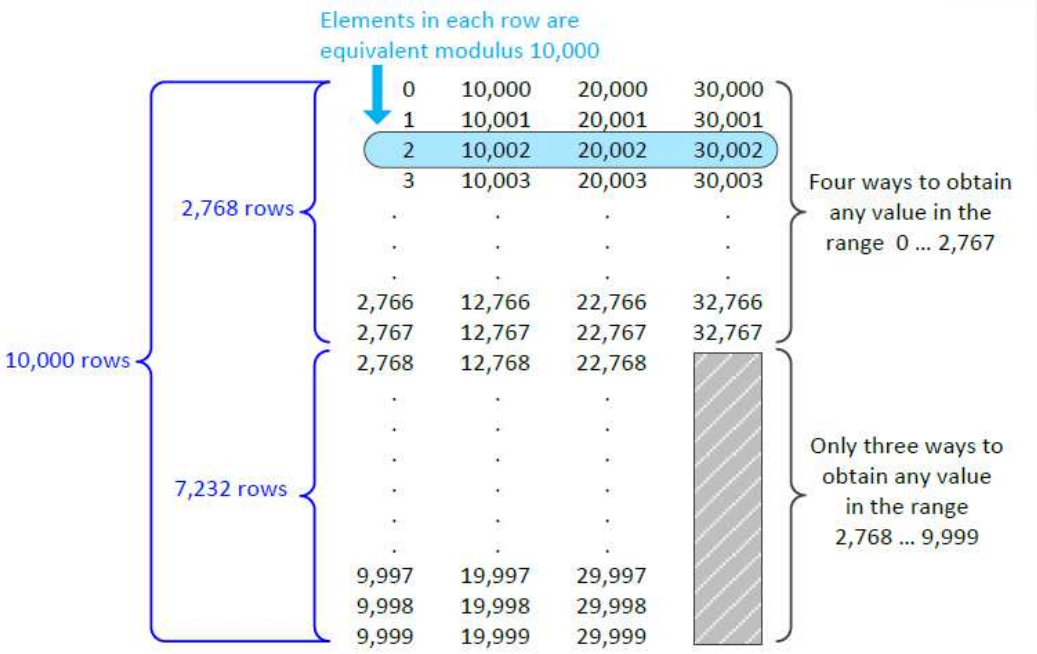
```
#include <iostream>
#include <string>
#include <sstream>

int main() {
   std::stringstream ss;

   for (int i = 0; i < 10; ++i)
      ss << i << ", ";

   std::cout << ss.str() << std::endl;
}
```


Better Pseudorandom Number Gen. (1)



Better Pseudorandom Number Gen. (2)

```
1: 175
2: 400
3: 17869
4: 30056
5: 16083
6: 12879
7: 8016
8: 7644
9: 15809
10: 1769
2147483649: 175
2147483650: 400
2147483651: 17869
2147483652: 30056
2147483653: 16083
2147483654: 12879
2147483655: 8016
2147483656: 7644
2147483657: 15809
2147483658: 1769
```


Better Pseudorandom Number Gen. (3)

- Mersenne twister algorithm
  - long period, $2^{19,937}$-1, which is approximately $4.3154 \times 10^{6,001}$
  - Mersenne prime: 2^n-1
  - MT19937: 32bit
  - MT19937-64: 64bit
- Seed
  - random_device
- Generator
  - mt19937, mt19937_64, default_random_engine
- Distribution
  - uniform_int_distribution, uniform_real_distribution, normal_distribution
- https://www.cplusplus.com/reference/random/

Better Pseudorandom Number Gen. (4)

```
#include <iostream>
#include <iomanip>
#include <random>
int main() {
   std::random_device rdev;
   std::mt19937 mt(rdev());
   std::uniform_int_distribution<int> dist(0, 99);
   // std::normal_distribution<double> dist(50., 10.);
   int histogram[100] = {0};
   for (int i = 0; i < 1000000; i++) {
      int r = dist(mt);
      // if(r >= 0 && r <= 99)
      histogram[r]++;
   }
   for (int i = 0; i < 100; i++)
      std::cout << i << ": " << histogram[i] << std::endl;
}
```


Object Basics

- In computer science, an object can be a variable, a data structure, a function, or a method, and as such, is a value in memory referenced by an identifier.
- In the object-oriented programming paradigm, object can be a combination of variables, functions, and data structures; in particular in class-based variations of the paradigm it refers to a particular instance of a class.
- An object is an instance of a class. The terms object and instance may be used interchangeably.

Instance Variables (1)

```
class className {
public: // access specifier
   type memberName;    // member
};

class Point {
public:
   double x;
   double y;
};

Point p1;       // instance, object
p1.x = 10;      // dot member selection operator(.)
p1.y = 20;
```


Instance Variables (2)

```
#include <iostream>
class Point {
public:
   double x; // The point's x coordinate
   double y; // The point's y coordinate
};
int main() {
   Point pt1, pt2;
   pt1.x = 8.5;        pt1.y = 0.0;
   pt2.x = -4;         pt2.y = 2.5;
   std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
   std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";
   pt1 = pt2;
   std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
   std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";
   pt1.x = 0;
   std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
   std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";
}
```

Instance Variables (3)

```
#include <iostream>
#include <string>
#include <vector>
class Account {
public:
   std::string name;
   int id;
   double balance;
};
void add_account(std::vector<Account>& accts) {
   std::string name;      int number;
   double amount;
   std::cout << "Enter name, account number, and account balance: ";
   std::cin >> name >> number >> amount;
   Account acct;
   acct.name = name;      acct.id = number;
   acct.balance = amount;
   accts.push_back(acct);
}
```


Instance Variables (4)

```
void print_accounts(const std::vector<Account>& accts) {
   int n = accts.size();
   for (int i = 0; i < n; i++)
      std::cout << accts[i].name << "," << accts[i].id
         << "," << accts[i].balance << '\n';
}

int main() {
   std::vector<Account> customers;
   add_account(customers);
   add_account(customers);
   print_accounts(customers);
}

//void swap(Account& er1, Account& er2) {
//   Account temp = er1;
//   er1 = er2;
//   er2 = temp;
//}
```

Member Functions (1)

```
std::vector<Account> accountDB;


bool withdraw(Account& acct, double amt) {
  bool result = false;
  if (acct.balance - amt >= 0) {
      acct.balance -= amt;
      result = true;
   }
   return result;
}
…
withdraw(accountDB[i], 10000.));
```


Member Functions (2)

```
class Account {
   string name;
   int id;
   double balance;
public:
   void deposit(double amt) {
      balance += amt;
   }
   bool withdraw(double amt) {
      bool result = false;
      if (balance - amt >= 0) {
         balance -= amt;
         result = true;
      }
      return result;
   }
};
```

public: directly referenced outside of the class
private: can be accessed only by functions within the class, default

```
Account acct;
…
acct.withdraw(100.);
```

Member Functions (3)

```
#include <iostream>
class Counter {
   int count;
public:
   void clear() { count = 0; }
   void inc() { count++; }
   int get() { return count; }
};

int main() {
   Counter ctr1, ctr2; // Declare a couple of Counter objects
   ctr1.clear();       // Reset the counters to zero
   ctr2.clear();
   ctr1.inc();
   std::cout << ctr1.get() << std::endl;
}
```


Constructors (1)

```
// Constructors are called when an instance of a class is created.
// A constructor has the same name as the class.
// A constructor has no return type (not even void).
#include <iostream>
#include <iomanip>
#include <string>
class Account {
   std::string name;             int id;
   double balance;
public:
   Account(const std::string& customer_name, int account_number,
   double amount): name(customer_name), id(account_number),
      balance(amount)
   {
      if (amount < 0) {
         std::cout << "Warning: negative account balance\n";
         balance = 0.0;
      }
   }
```

Constructors (2)

```
   void deposit(double amt) {
      balance += amt;
   }
   bool withdraw(double amt) {
      bool result = false;
      if (balance - amt >= 0) {
         balance -= amt;
         result = true;
      }
      return result;
   }
   void display() {
      std::cout << "Name: " << name << ", ID: " << id
         << ", Balance: " << balance << '\n';
   }
};
```


Constructors (3)

```
int main() {
   Account acct1("Joe", 2312, 1000.00);
   Account acct2("Moe", 2313, 500.29);
   acct1.display();              acct2.display();
   std::cout << "---------------------" << '\n';
   acct1.withdraw(800.00);
   acct2.deposit(22.00);
   acct1.display();              acct2.display();
}
// constructor initialization list ":" , reference or const member

// A class must have at least one constructor, either defined by the
// program or by the compiler.
// The default constructor is called whenever an object is created
// without passing any arguments.
// If you define any type of constructor, you must also define a
// default constructor if it is needed.
```

Uniform initialization

```
T()
T{} // uniform (braces) initialization

int a(10), b(10.5), c{20}, d{20.5};

class A { … };
A a1(1), a2(), a3{}, a4{1}, a5{1, 2}; // a2: function or instance?

class B {
public:
   std::vector<int> v;
   B(std::initializer_list<int> l) : v(l) {}
   …
};

B b1{1, 2, 3, 4, 5};
```


Defining a New Numeric Type (1)

```
#include <iostream>
#include <cstdlib>
class SimpleRational {
   int numerator;
   int denominator;
public:
   SimpleRational(int n, int d): numerator(n), denominator(d) {
      if (d == 0) {
         std::cout << "Zero denominator error\n";
         exit(1);
      }
   }
   SimpleRational(): numerator(0), denominator(1) {}
   void set_numerator(int n) {
      numerator = n;
   }
```

Defining a New Numeric Type (2)

```
   void set_denominator(int d) {
      if (d != 0) denominator = d;
      else {
         std::cout << "Zero denominator error\n";
         exit(1);
      }
   }
   int get_numerator() {      return numerator;   }
   int get_denominator() {    return denominator; }
};
SimpleRational multiply(SimpleRational f1, SimpleRational f2) {
   return {f1.get_numerator() * f2.get_numerator(),
      f1.get_denominator() * f2.get_denominator()};
}
void print_fraction(SimpleRational f) {
   std::cout << f.get_numerator() << "/" << f.get_denominator();
}
```


Defining a New Numeric Type (3)

```
int main() {
   SimpleRational fract(1, 2); // The fraction 1/2
   std::cout << "The fraction is ";
   print_fraction(fract);
   std::cout << '\n';
   fract.set_numerator(19);
   fract.set_denominator(4);
   std::cout << "The fraction now is ";      print_fraction(fract);
   std::cout << '\n';

   SimpleRational fract1{1, 2}, fract2{2, 3};
   auto prod = multiply(fract1, fract2);
   std::cout << "The product of ";    print_fraction(fract1);
   std::cout << " and ";              print_fraction(fract2);
   std::cout << " is ";               print_fraction(prod);
   std::cout << '\n';
}
```

Encapsulation

- The class interface—the visible part.
  - Clients see and can use the public parts of an object.
  - The public methods and public variables of a class constitute the interface of the class.

- The class implementation—the hidden part.
  - Clients cannot see any private methods or private variables. Since this private information is invisible to clients, class developers are free to do whatever they want with the private parts of the class.