05. A Tour of C++: Modularity

Data Structure and Algorithms

Hanjun Kim
Compiler **O**ptimization **Re**search **Lab**
Yonsei University

Last Time: Memory

- Pointers
- References
- Pointers and Arrays
  - Arrays
  - Pointer Arithmetics
- Dynamic Allocation
- Pointers and const
- Function Pointers
- Memory Layout

Today: Modularity

- Function
  - Function
  - Call by Value vs. Call by Reference
  - Template
- Namespace
  - Scope
  - Namespace
- Programmer-defined types: Structures
- Programmer-defined types: Classes

Modularity

- Modularity
  - Separates the functionality of a program into independent, interchangeable modules
  - Distinguishes between the interface (declaration) to a part and its implementation (definitions)
  - Declaration
    - Introduces a name into a scope
    - Specifies all that's needed to use a function or a type

```
double sqrt(double); // the square root function
                     // takes a double and returns a double
```

  - Definition
    - Fully specifies the declared entity (Function body)
    - Its representation is "elsewhere"

```
double sqrt(double d) // definition of sqrt()
{
  // ... algorithm as found in math textbook ...
}
```

Function

- Function
  - A group of statements that together perform a task
  - General form:
    - Declaration

```
return_type name (formal arguments);
```

    - Definition

```
return_type name (formal arguments) body
```

    - A body is a block or a try block

```
// a block
{/* code */}
```

```
try { /* code */ } // a try block
catch(exception& e) { /* code */ }
```

    - For example

```
double f(int a, double d) { return a*d; }
```

  - Allows to chop a program into manageable pieces
    - Divide and Conquer
    - Ease testing, distribution of labor, and maintenance

Function

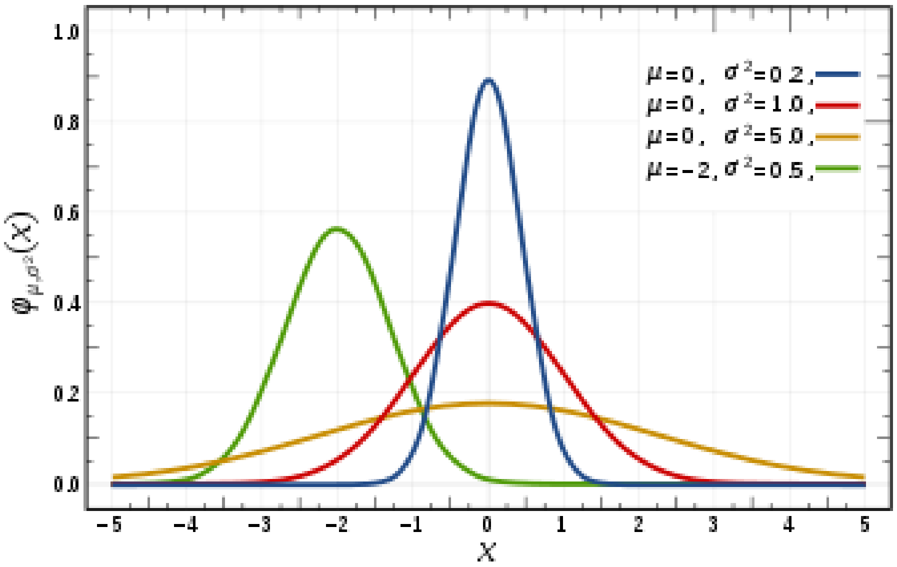
- Functions provide a new way to control the flow of execution.

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

double mySqrt(double n) {
  double error = 1E-5;
  if (n < 0) return -1;
  double t = n;
  while (abs(t - n/t) > error) t = (t+n/t)/2.0;
  return t;
}

int main (){
  double n, t;
  cin >> n;
  t = mySqrt(n);
  cout << "Sqrt of " << n << " is " << t << "!\n";
}
```

sqrt3.cpp

Function Example

- Standard Gaussian distribution: Bell curve
  - Basis of most statistical analysis in social and physical sciences
  - $\phi(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
  - Use built-in functions when possible; build your own when not available



The red curve is the *standard normal distribution*

Function Example

- Standard Gaussian distribution: Bell curve
  - `phi(double x)` and `phi(double x, double mu, double sigma)` are different (Overloading)

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

double phi(double x) {
  return exp(-x*x / 2)/sqrt(2 * M_PI);
}

double phi(double x, double mu, double sigma){
  return phi((x - mu) / sigma) / sigma;
}

int main (){
  double x, mu, sigma, pdf;
  cin >> x >> mu >> sigma;
  pdf = phi(x, mu, sigma);
  cout << "probability density is " << pdf << "!\n";
}
```

gaussian.cpp

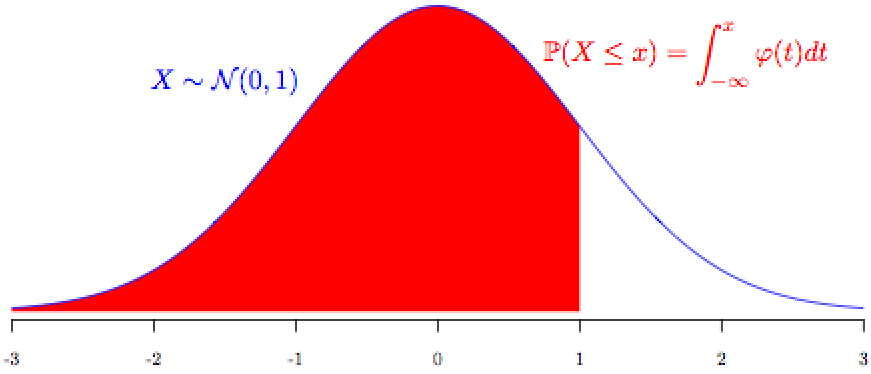
Overloading: Functions with different signatures are different

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

$$\phi(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Function Example

- Standard Gaussian distribution: Bell curve
  - Compute Gaussian cdf $\Phi(z)$
  - Use Taylor series for cdf calculation
  - $\Phi(z) = \int_{-\infty}^{z} \phi(x)dx = \frac{1}{2} + \phi(z)\{z + \frac{z^3}{1 \cdot 3} + \frac{z^5}{1 \cdot 3 \cdot 5} + \frac{z^7}{1 \cdot 3 \cdot 5 \cdot 7} + \cdots\}$



Function Example

- Standard Gaussian distribution: Bell curve
  - Compute Gaussian cdf $\Phi(z)$
  - Use Taylor series for cdf calculation

```
...

double Phi(double z) {
  if (z < -8.0) return 0.0;
  if (z >  8.0) return 1.0;
  double sum = 0.0, term = z;
  for (int i = 3; sum + term != sum; i += 2) {
    sum  = sum + term;
    term = term * z * z / i;
  }
  return 0.5 + sum * phi(z);
}

double Phi(double z, double mu, double sigma){
  return Phi((z - mu) / sigma);
}
                                   gaussian.cpp
```

Function: Call by Value vs. Call by Reference

- Call by Value
  - Send the function a copy of the argument’s value
  - Example

```
void byValue(int x, int y);
```

1 3

```
z  = byValue(  1  ,  3   );
```

```
void byValue() {
  int x = 1;
  int y = 3;
  ...
}
```

- Call by Reference
  - Pass a reference to the argument (Aliasing)

```
void byRef(int& x, int& y);
```

a b

```
z  = byRef(  a  ,  b   );
```

```
void byRef() {
  int& x = a;
  int& y = b;
  ...
}
```

Function: Call by Value vs. Call by Reference

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

void callByValueNRef(int a, int& b){
  a*=2;
  b*=2;
}

int main () {
  int x=1, y=3;
  cout << "x=" << x << ", y=" << y << endl;
  callByValueNRef(x, y);
  cout << "x=" << x << ", y=" << y << endl;
  return 0;
}                                           callByValueNRef.cpp
```

```
x=1, y=3
x=1, y=6
```

- `a`: Call by Value
  - Value of `x` (1) is copied to `a`. `x` is not changed
- `b`: Call by Reference
  - `b` and `y` are aliased. `y` becomes doubled

Overload

- Overloaded functions
  - Functions that have the same name but different parameters
  - Compilers know which one to call using types of the parameters

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

int operate (int a, int b){  return (a*b); }

double operate (double a, double b){  return (a/b); }

int main ()
{
  int x=5, y=2;
  double n=5.0, m=2.0;
  cout << operate (x, y) << '\n';
  cout << operate (n, m) << '\n';
  return 0;
}
```

```
10
2.5
```

overload.cpp

Template

- Motivation
  - Overloaded functions may have the same definition (same body)
  - Need to define functions with generic types (template)

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

int sum(int a, int b){
  return a+b;
}

double sum(double a, double b){
  return a+b;
}

int main () {
  cout << sum(3, 4) << '\n';
  cout << sum(3.1, 4.2) << '\n';
  return 0;
}
```

```
7
7.3
```

overload2.cpp

Template

- Template
  - The same syntax as a regular function except `template` keyword

```
template <template-parameters>
function-declaration
```

  - Usage

```
function_name <template-arguments> (function-arguments)
```

  - Example

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

template <class SomeType>
SomeType sum (SomeType a, SomeType b) {
  return a+b;
}

int main () {
  cout << sum<int>(3, 4) << '\n';
  cout << sum<double>(3.1, 4.2) << '\n';
  return 0;
}
```

```
7
7.3
```

template.cpp

Template: Another Example

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

template <class T, class U>
bool are_equal (T a, U b) {
  return (a==b);
}

int main () {
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

```
x and y are equal
```

template2.cpp

- `are_equal`: Template with multiple parameters
  - Parameters: Class `T` and class `U`
- `are_equal(10,10.0)`: Usage with type inference
  - `are_equal<int,double>(10,10.0)`

Template

- Non-type template arguments
  - The template parameters can include expressions of a particular type
  - Similar to a regular function parameter, but the parameter value is determined on compile-time (the value of that argument is never passed during runtime)

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

template <class T, int N>
T fixed_multiply (T val) {
  return val * N;
}

int main() {
  cout << fixed_multiply<int,2>(10) << '\n';
  cout << fixed_multiply<int,3>(10) << '\n';
}
```

```
20
30
```

template3.cpp

Scope

- Scope: A region of program text
  - Global scope: outside any language construct
  - Class scope: within a class
  - Local scope: between { … } braces
  - Statement scope: in a statement (e.g. in a for-statement)
- A name in a scope
  - Seen within its scope and its nested scope
  - Only one name in a scope
  - Only after its declaration (“can’t look ahead” rule)
    - Exception: Class members can be used within the class before their declaration

Scope

```
#include <cmath>                    // max, abs: global scope

// no r, i, or v here
class My_vector {
  int v[100];                       // v: class scope
public:
  int largest()                     // largest: class scope
  {
    int r = 0;                      // r: local scope
    for (int i = 0; i<100; ++i)     // i: statement scope
      r = max(r,abs(v[i]));
    // no i here
    return r;
  }
  // no r here
};
// no v here
```

Nested Scope

```
int x; // global variable - avoid those where you can
int y; // another global variable

int f()
{
  int x;  // local variable (Note - now there are two x's)
  x = 7;  // local x, not the global x
  {
    int x = y;  // another local x, initialized by y
                // (Now there are three x's)
    ++x;        // increment the local x in this scope
  }
}

// avoid such complicated nesting and hiding: keep it simple!
```

Namespace

- Motivation
  - Only one entity can exist with a particular name in a particular scope
  - Possible name collision for non-local names
- Namespace
  - Introduces a namespace scope
  - Changes names from global scopes to narrower namespace scopes
  - Declaration

```
namespace identifier {
  named_entities
}
```

```
namespace myNamespace{
  int a, b;
}
```

  - Usage

```
namespace_id::named_entities
```

```
myNamespace::a
myNamespace::b
```

Namespace Example

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

namespace foo {
  int value() { return 5; }
}

namespace bar {
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

namespace.cpp

Namespace: `using namesapce`

- `using namespace`
  - Introduces a name into the current declarative region
  - Avoids the need to qualify the name
  - Example
    - `Value` can be used without name qualifiers `foo` and `bar`

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

namespace foo {
  int value() { return 5; }
}

namespace bar {
  const double pi = 3.1416;
  double value() { return 2*pi; }
}
```

```
int main () {
  {
    using namespace foo;
    cout << value() << '\n';
  }
  {
    using namespace bar;
    cout << value() << '\n';
    cout << pi << '\n';
  }
}
```

namespace2.cpp

Programmer-defined types: Structures

- Structure
  - A collection of variables of different data types under a single name
  - `struct` keyword defines a structure type
  - Example: Vector
    - Two members: sz and elem

```
struct Vector {
  int sz; // number of elements
  double* elem; // pointer to elements
};
```

  - Defining a structure variable
    - Allocates its required memory

```
Vector v;
```

  - Accessing a member of a structure variable
    - Use a dot (.) operator

```
v.sz = s;
```

Programmer-defined types: Structures

- Structure Example: Vector

```
struct Vector {
  int sz; // number of elements
  double* elem; // pointer to elements
};

void vector_sum() {
  Vector v;
  int s = 0;
  cout << "Enter the size of array: ";
  cin >> s;
  v.elem = new double[s];
  v.sz = s;
  for (int i=0; i!=s; ++i) {
    cout << "Enter value of the element "<<i+1 << ":";
    cin>>v.elem[i];
  }
  double sum = 0;
  for (int i=0; i!=s; ++i) sum+=v.elem[i];
  cout << "Their sum is " << sum << endl;
}
```

vector.cpp

Programmer-defined types: Structures

- Pointers to structures
  - Use the arrow operator (->) to access a member of a structure variable

```
void vector_pointer_sum() {
  Vector v;
  Vector *pV = &v;
  int s = 0;
  cout << "Enter the size of array: "; cin >> s;
  pV->elem = new double[s];
  pV->sz = s;
  for (int i=0; i!=s; ++i) {
    cout << "Enter value of the element "<<i+1 << ":";
    cin>>pV->elem[i];
  }
  double sum = 0;
  for (int i=0; i!=s; ++i) sum+=pV->elem[i];
  cout << "Their sum is " << sum << endl;
}
```

vector.cpp

Programmer-defined types: Classes

- Class
  - An expanded concept of data structures
  - Like structures, classes can contain a collection of variables
  - Unlike structures (C style), classes can contain functions as members
  - Class object: An instantiation of a class in memory
  - Class declaration

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
};
```

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
};
```

  - Class definition

```
class_name object_name;
```

```
Rectangle rect;
```

Programmer-defined types: Classes

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

- Class
  - class_name
    - An identifier for the class
  - object_names (Optional)
    - Objects of this class
  - Members: data or function declarations
  - Access specifiers: Access rights for the members
    - `private`: accessible only from within the same class (including "friends")
    - `protected`: accessible from the same and derived class (including "friends")
    - `public`: accessible from anywhere where the object is visible.
    - Default: `private`
  - The scope operator (::) specifies the class to which the member being defined belongs

Programmer-defined types: Classes

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

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area () {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

rectangle.cpp

Programmer-defined types: Classes

- Constructors
  - Automatically called whenever a new object of this class is created
  - Functions with the class name and without any return type

```
class_name();                // default constructor
```

```
class_name(parameters);   // constructor with paramemters
```

  - Constructor overloading
    - Constructors with different parameters
- Destructors
  - Automatically called when an object is destructed
  - Execute required cleanup
  - A member function with the class name preceded with a tilde sign (~), and without any return type

```
~class_name();               // default destructor
```

Programmer-defined types: Classes

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

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

rectangle2.cpp

Programmer-defined types: Classes

- Pointers to classes
  - Use the arrow operator (->) to access a class member

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

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle* rect = new Rectangle(3,4);
  cout << "rect area: " << rect->area() << endl;
  return 0;
}
```

```
rect area: 12
```

rectangle3.cpp

Programmer-defined types: Classes

- Operator overloading
  - Overloading operations on a certain class type

```
type operator sign (parameters) { /*... body ...*/ }
```

  - Overloadable operators

```
 +    -    *    /    =    <    >   +=   -=   *=   /=    <<   >>
<<=   >>=  ==   !=   <=   >=   ++   --    %    &    ^     !    |
 ~     &=  ^=   |=   &&   ||   %=   []   ()    ,   ->*   ->
new   delete   new[]    delete[]
```

  - Example

```
CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}
```

Programmer-defined types: Classes

- Operator overloading
  - Operators and member functions (Here, @ is an operator)

<table>
<tr><th>Expression</th><th>Operators</th><th>Member Function</th><th>Non-member Function</th></tr>
<tr><td>@a</td><td>+ - * & ! ~ ++ --</td><td>A::operator@()</td><td>operator@(A)</td></tr>
<tr><td>a@</td><td>++ --</td><td>A::operator@(int)</td><td>operator@(A, int)</td></tr>
<tr><td>a@b</td><td>+ - * / % ^ & | < >
== != <= >= << >> && || ,</td><td>A::operator@(B)</td><td>operator@(A, B)</td></tr>
<tr><td>a@b</td><td>= += -= *= /= %= ^= &=
|= <<= >>= []</td><td>A::operator@(B)</td><td>-</td></tr>
<tr><td>a(b,c,…)</td><td>()</td><td>A::operator()(B,C, …)</td><td>-</td></tr>
<tr><td>a->b</td><td>-></td><td>A::operator->()</td><td>-</td></tr>
<tr><td>(TYPE) a</td><td>TYPE</td><td>A::operator TYPE()</td><td>-</td></tr>
</table>

Programmer-defined types: Classes

```
class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

CVector operator- (const CVector& lhs, const CVector& rhs) {
  CVector temp;
  temp.x = lhs.x - rhs.x;
  temp.y = lhs.y - rhs.y;
  return temp;
}
```

operators.cpp

Programmer-defined types: Classes

```
int main () {
  CVector foo (3,1), bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  result = foo - bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}                                        operators.cpp
```

```
4,3
2,-1
```

- `operator+`
  - Overloading + operator with a member function (a member of Cvector class)
- `operator-`
  - Overloading - operator with a non-member function

Programmer-defined types: Classes

- Class Template
  - Allows classes to have members that use template parameters as types
  - Example

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second) {
      values[0]=first;
      values[1]=second;
    }
};
```

template4.cpp

```
mypair<int> myobject (115, 36);
mypair<double> myfloats (3.0, 2.18);
```

Programmer-defined types: Classes

- Template Specialization
  - Defines a different implementation for a template when a specific type is passed as template argument
  - Example

```
template <>
class mypair <char> {
    string values;
  public:
    mypair (char first, char second) {
      values+=first;
      values+=second;
    }
    void concatenate(char c) {
      values+=c;
    }
};
```

template4.cpp

```
mypair<char> mystring ('a', 'b');
mystring.concatenate('c');
```

Programmer-defined types: Classes

- Special members
  - `this`
    - A pointer to the object whose member function is being executed
    - Used to refer to the object itself within a class's member function
  - Static member variable
    - Only one common variable for all the objects of that same class
    - Shares the same value
  - Static member function
    - A member of a class that are common to all object of that class
    - Acting exactly as non-member functions

Programmer-defined types: Classes

- Special Constructors and Assignments
  - Default constructor: initialized without any argument
  - Copy constructor: A constructor whose first parameter is of type reference to the class itself

```
MyClass::MyClass (const MyClass&);
```

  - Copy assignment operator: An overload of operator= which takes a value or reference of the class itself as parameter

```
MyClass& operator= (const MyClass&);
```

  - Move constructor: called when initialized using an unnamed temporary

```
MyClass (MyClass&&);
```

  - Move assignment operator: An overload of operator= taking an unnamed class

```
MyClass& operator= (const MyClass&&);
```

```
MyClass foo;                // default constructor
MyClass bar = foo;          // copy constructor
foo = bar;                  // copy assignment
MyClass baz = fn();         // move constructor
baz = MyClass();            // move assignment
```

Structure in C, C++ and Class in C++

- Structure in C
  - Can support only public member variables
  - No member function, access specifier, no inheritance

- Structure in C++
  - Can support member variables and member functions
  - Can have access specifiers (default: public)
  - Can support inheritance

- Class in C++
  - Can support member variables and member functions
  - Can have access specifiers (default: private)
  - Can support inheritance

Summary: Modularity

- Function
  - Function
  - Call by Value vs. Call by Reference
  - Template
- Namespace
  - Scope
  - Namespace
- Programmer-defined types: Structures
- Programmer-defined types: Classes