Programming in C++

Balázs Goldschmidt

Introduction

- C was extremely successful
  - source available
  - mother tongue of UNIX
- Object orientation helps solving the software crisis
  - data and operations encapsulated
  - inheritance provides reusability
  - polymorphism provides flexibility
  - closer to real world entities

Introduction 2

- Adding OO to C was a goal of many
  - C
    - OO like libraries (X-window, Motif, etc.)
  - Objective-C
    - classes, interfaces, extended syntax
  - C++ *(Bjarne Stroustrup)*
    - classes, extended syntax, as close to C as possible
    - main goal: compatibility and OO

Non Object Oriented Features

Comments

- C style comments
  - `/* ... */`
  - can not be nested
- Line comments
  - `// ...`
  - ends when line ends
  - can be put inside a C style comment
  - e.g.: `int x = 10; // initial value for 10`

Variable declaration

- C
  - variable declaration
    - global, local, or formal parameter
    - local variables declared only at beginning of function
  - extern variables
    - declared in file
    - defined elsewhere
    - common variables among different modules/files

Variable declaration 2

- C++
  - variable declaration
    - global, local, or formal parameter
    - local variables declared anywhere
    - best practice: declare near to its use
  - extern variables
    - same as in C

Variable declaration 4

```
void foo() {
...
int x = doit();
...
x++;
int y=2*x;
for (int i = 0; i<x; i++)
   {...}
...
}
```


Function overload

- In C a function's name defines the function
  - for different types something, like the typename must be included
  - for example maximum calculation:

```
int maxInt(int a, int b) {
      return (a<b)?b:a;
}

double maxDouble(double a, double b) {
      return (a<b)?b:a;
}
```

Function overload 2

- In C++ the function is identified by its name *and parameters*
  - return value doesn't count

```
int max(int a, int b) {
      return (a<b)?b:a;
}

double max(double a, double b) {
      return (a<b)?b:a;
}
```


Default parameters

- In C++ function parameters can have a default value
  - only last parameters
  - when called, omitted parameters have the predefined value

```
char* conv(int n, int from = 10, int to = 2) {
      ...
}

char* x = conv(123, 8, 10); // 123(8) -> 83(10)
char* y = conv(123, 8);     // 123(8) -> 1010011(2)
char* z = conv(123);  // 123(10)-> 1111011(2)
```


Inline functions

- In C functions vs. macros
  - function
    - heavy weight
      - stack handling, parameter passing
      - call is compiled in
    - strict typing
  - macro
    - lightweight
      - no stack, no parameter passing
      - directly compiled into code
    - no types

Inline functions 2

- Function disadvantages
  - overhead
- Macro disadvatages
  - side effects multiplied

```
#define max(a,b) ((a)<(b)?(b):(a))

int x = 3;
int y = 6;

int z = max(x++, y++);
// x = 4, y = 7, z = 6
```

```
int z = ((x++)<(y++)?(y++):(x++));
// x = 4, y = 8, z = 7
```


Inline functions 3

- C++ solution: inline functions
  - works like a function
  - compiled directly into code

```
inline int max(int a,int b) {
      return ((a)<(b)?(b):(a));
}

int x = 3;
int y = 6;
int z = max(x++, y++);
// x = 4, y = 7, z = 6
```


Inline functions 4

- Disadvantages
  - function for each type
    - solved later by templates
  - should be done by optimizer not programmer

Reference types

- C parameter passing: by value
  - using pointers for side-effects
    - e.g. `scanf("%d", &i)`
  - disadvantage: easy to forget
- C++ introduces references
  - reference: new name for a variable

```
int i = 10;
int& j = i;
j++;
printf("%d %d\n", i, j);  // 11, 11
```


Reference types 2

- Reference is always initialized
- Works even in function calls

```
void foo(int& i, int j) {
        i++;
        j++;
}
```

```
int a = 10;
int b = 20;
foo(a, b);
printf("%d %d\n", a, b); // 11, 20
```


Reference types 3

- Dangers and annoyances
  - client developer might forget side-effects
  - function developer might use it to often
    - less memory-copy
    - results unkown side-effects
  - conversion to pointer is dangerous
    - `int* foo(int& i) { return &i; }`
  - in many cases use of pointers is advised
    - especially if some parts can not be done without pointers

Constant types

- In K&R C all variables are *variable*
  - constants are implemented as macros
  - `#define PI 3.14159265358979323844`
- C++ introduces constants
  - *const* modifier
  - must be initialized

Constant types 2

- `int i0 = 10;`
- `const int i1 = i0 + 2;`
  - the value of *i1* cannot be modified
  - `i1++; // error`
- `const int* i2 = &i0;`
  - the value pointed by *i2* cannot be modified
  - the value of the pointer can be modified
  - `(*i2)++; // error`
  - `i2++;  // OK`

Constant types 3

- `int i0 = 10;`
- `int* const i3 = &i0;`
  - the value pointed by *i3* can be modified
  - the value of the pointer cannot be modified
  - `(*i3)++; // OK`
  - `i3++;  // error`
- `const int* const i4 = &i0;`
  - the value pointed by *i4* cannot be modified
  - the value of the pointer cannot be modified
  - `(*i4)++; // error`
  - `i4++;  // error`

Struct and enum name is typename

- In C referring to a struct or enum needs also the struct or enum keyword

```
struct foo { int d; };
struct foo* f =
      (struct foo*)malloc(3*sizeof(struct foo));
```

- Typedefs can help

```
typedef struct { int d; } bar;
bar* f = (bar*)malloc(3*sizeof(bar));
```


Struct name is typename 2

- In C++ referring to a struct or enum only needs the name of the struct or enum

```
struct foo { int d; };
foo* f = (foo*)malloc(3*sizeof(foo));

enum day { monday, tuesday, ...};
int addEntry(day d, ...);
```


New type: bool

- works like a correct boolean implementation
  - literals: *true* and *false*
- can be mixed with integers, but are not integers
  - integer value always 1 or 0

```
bool a = true;
a++;
a = a+5;
int x = a+3;
bool a2 = x-4;
printf("%d %d %d\n", a, x, a2);
// 1 4 0
```

Memory handling

- Memory handling in C

```
int* ip = (int*)malloc(10*sizeof(int));
…
free(ip);
```

- Memory handling in C++

```
int* ip = new int[10];
…
delete [] ip;
```


Memory handling 2

- Problems with C
  - cumbersome, not OO
- In C++
  - new operators
    - new, delete
    - new[], delete[]
  - no realloc
  - OO compatible, easier

Defining Classes and Objects

Object-orientation intro

- Problem:

  Let's define a complex type in C with the following operations: addition and multiplication.

  New C++ features can be used.

Complex type in C

```
struct Complex {
       double re;
       double im;
};
Complex add(Complex* this, Complex c) {
       Complex r;
       r.re = this->re + c2.re;
       r.im = this->im + c2.im;
       return r;
}
Complex mult(Complex this, Complex c) {
       ...
}
```

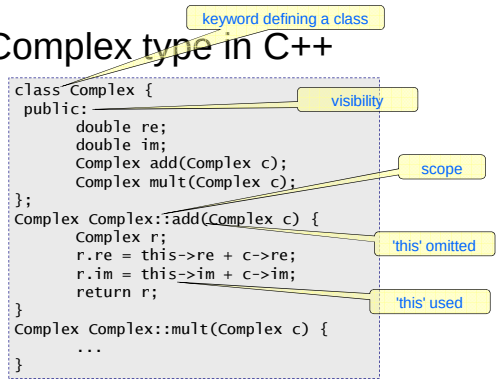

Complex type in C (2)

```
void main(void) {
       ...
       Complex c1, c2, c3;

       c1.re = 1;
       c1.im = 2;

       c2.re = 0;
       c2.im = 5;

       c3 = add(&c1, c2);
       ...
}
```

Complex type in C (3)

- Problems
  - initialization
    - should be in one step
  - data and operation separately
    - easy to modify just one without the other
  - unlimited access to members
    - modifications are not controlled

Complex type in C++




```
class Complex {
 public:
       double re;
       double im;
       Complex add(Complex c);
       Complex mult(Complex c);
};
Complex Complex::add(Complex c) {
       Complex r;
       r.re = this->re + c->re;
       r.im = this->im + c->im;
       return r;
}
Complex Complex::mult(Complex c) {
       ...
}
```


Complex type in C++ (2)

```
void main(void) {
       ...
       Complex c1, c2, c3;
       c1.re = 1; c1.im = 2;
       c2.re = 0; c2.im = 5;
       c3 = c1.add(c2);
       ...
}
```

method invocation
this = &c1

Visibility

- public
  - can be accessed from anywhere
- protected
  - can be accessed by subclasses and the class only
- private
  - can be accessed by the class only

Method definition

- Outside definition
  - declaration in class body
  - definition outside with scopes
  - like in previous example
- Inline definition
  - declaration and definition in class body
  - always compiled inline

Method definition 2

- Inline definition

```
class Complex {
      ...
      Complex add(Complex c){
            Complex r;
            r.re = this->re + c->re;
            r.im = this->im + c->im;
            return r;
      }
      ...
};
```

Initialization: constructors

- The initialization is still done manually
  - attributes can not be private
- Solution: constructor
  - called when object is created
  - similar to a method
  - name is classname
  - no return value
  - default constructor only if there is none defined

Initialization: constructors 2

```
class Complex {
       double re;
       double im;
public:
       Complex(double d1=0, double d2=0) {
              re = d1; im = d2;
       }
       ...
};
```

private by default

default parameter

```
void main(void) {
       ...
       Complex c1(1,2), c2(5), c3;
       c3 = c1.add(c2);
       ...
}
```


Problem #2

- Dynamic integer array

Let's create an array that can store integers. If a new integer is added to the array, its size should grow.

The memory management should be correct.

Dynamic integer array

```
class IntArray {
      int* array; // the integers
      int size;   // size of the array
  public:
      IntArray(); // ctr
      void add(int i); // appends i to the end
      int& get(int index); // return int
                           // at index
      int getSize();       // returns size

      ???
};
```

can be left value

read only attribute

Dynamic integer array 2

```
IntArray::IntArray() {
      size = 0;
      array = new int[0];
}

void IntArray::add(int d) {
      int* tmp = new int[size+1];
      for (int i = 0; i < size; i++)
            tmp[i] = array[i];
      // memcpy(tmp, array, size*sizeof(int));
      tmp[size] = d;
      delete [] array;
      array = tmp;
      size++;
}
```


Dynamic integer array 3

```
int& IntArray::get(int index) {
      if (index < 0) || (index >= size) {
            // problem
      }
      return array[index];
}

int IntArray::getSize() {
      return size;
}
```

Dynamic integer array 4

```
int main() {
      ...
      IntArray ia;

      ia.add(10);
      ia.add(4);
      ia.add(12);

      ia.get(2) = 6;

      for (int i = 0; i < ia.getSize(); i++) {
            printf("%d\n", ia.get(i));
      }
      ...
}
```


Destructors

- What is missing from *IntArray*?
  - who deletes the *array* pointer?
  - **destructors** have to be defined
  - destructor called when object is deleted either explicitly or implicitly

```
class IntArray {
      ...
  public:
      ...
      ~IntArray() { delete [] array; }
};
```


Destructors and delete

- New and Delete can be used for allocating dynamic memory
  - *new* calls constructor
  - *delete* calls destructor
- In case of arrays
  - `Complex* p = new Complex[10]` creates an array of 10 complex numbers
  - `delete p` deletes array, but only calls destructor on first element
  - `delete[] p` deletes array, and calls destructor on each element

Copy constructor

- What happens to objects passed as parameters?

```
IntArray foo(IntArray ia) {
      IntArray ret;
      ret.add(ia.getSize());
      ia.add(10);
      return ret;
}
```

ia	**4;5**;10
ret	2
[return]	2

```
IntArray ar;
ar.add(4); ar.add(5);
IntArray ar2 = foo(ar);
```

ar	4;5
ar2	2

Copy constructor 2

- Default passing behaviour:
  - bitwise copy
  - for object fields copy constructor is called
- To modify define copy constructor
  - like a normal constructor
  - single parameter with type `const X&`
- Copy cosntructor is called
  - when passed as parameter
  - when returned
  - when temporary variables in expressions

Copy constructor 3

```
class IntArray {
      ...
  public:
      IntArray(const IntArray& ia) {
            size = ia.size;
            array = new int[size];
            for (int i = 0; i < size; i++) {
                  array[i] = ia.array[i];
            }
      }
      ...
};
```

Operator overloading

- Problem:

  Modify Complex class so that mathematical operators work!

Operator overloading 2

- operators:
  - complex + complex
  - complex + double
  - double + complex
  - multiplication, division, etc similarly

Operator overloading 3

```
class Complex {
      double re;
      double im;
  public:
      ...
      // complex + complex
      Complex operator+(Complex c) {
            Complex ret(re+c.re, im+c.im);
            return ret;
      }
      // complex + double
      Complex operator+(double d) {
            Complex ret(re+d, im);
            return ret;
      }
};
```

Operator overloading 4

```
int main() {
	Complex c1(3,4);
	Complex c2(5,6);
	Complex c3;

	c3 = c1+c2;
	// c3 = c1.operator+(c2) // complex

	c2 = c1 + 4.5;
	// c2 = c1.operator+(4.5) // double

}
```


Operator overloading overview

- Rules
  - only existing operators
    - except: `:: ?: sizeof .`
  - syntax, precedence, and number of operands cannot be modified
- Special operators
  - `operator++()` preincrement
  - `operator++(int)` postincrement
  - `operator double()` cast operator
  - `operator[](int)` index operator
  - `operator()()` function call operator

Operator overloading 5

```
int main() {
	Complex c1(3,4);
	Complex c2(5,6);
	Complex c3;

	c3 = c1+c2;
	// c3 = c1.operator+(c2) // complex

	c2 = c1 + 4.5;
	// c2 = c1.operator+(4.5) // double

	c3 += c1; // ???

	double d = 6.7 + c1; // ???
}
```

Operator overloading 6

- Do we have now *operator+=* ?
  - no, this is a totally different operator
  - won't be generated automatically
- What to do with *double* + *complex*?
  - ~~`double::operator+(Complex c)`~~
  - problem: *double* is not a class
  - solution: global operator

Friend

```
Complex operator+(double d, const Complex& c) {
            Complex ret(c.re+d, c.im);
            return ret;
}
```

- Problem: *re* and *im* are private
- Solutions:
  - getter methods
    - getRe(), getIm()
  - friend functions
    - extra right to access private members

Friend 2

```
class Complex {
      double re;
      double im;
  public:
      ...
      // double + complex
      friend Complex operator+(double d, Complex c);
};
```

- Can be inline or defined outside the function
- Usually used when first operand is
  - either primitive type (int, double, etc)
  - or unmodifiable class

Friend 3

- Typical example: stdio in C++
- New classes: `istream`, `ostream`
- Operators: `<<` and `>>` respectively
- New global variables: `cin`, `cout`, `cerr`
  - standard in, out and error streams respectively
- For primitive types predefined
- For classes programmer has to implement it:

```
istream& operator>>(istream& is, X& x);
ostream& operator<<(ostream& os, X& x);
```

allows cascade

Friend 4

```
#include <iostream>
using namespace std; // discussed later
class Complex {
    double re;
    double im;
  public:
      ...
    friend istream& operator>>(istream& is, Complex& c);
    friend ostream& operator<<(ostream& os, Complex& c){
       os << re << "+" << im << "i";
       return os;
};
istream& operator>>(istream& is, Complex& c) {
       is >> re; is >> im;
       return is;
}
```

cascade use

Friend 5

```
int main() {
       Complex c1, c2;

       // reading 2 complex numbers from stdin
       cin >> c1 >> c2;

       // alternatively with no cascade
       // cin >> c1;
       // cin >> c2;

       // writing sum to stdout
       // with closing newline and flushing
       cout << (c1+c2) << endl;
}
```

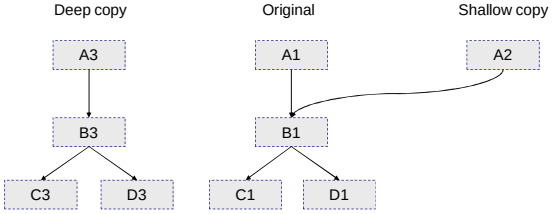
Operator overloading: index

```
class IntArray {
...
      int& operator[](int index) {
            if (index < 0 || index >= size) {
                  // error
            }
            return array[index];
      }
};
...
IntArray a;
a.add(3);
a.add(5);
a[0] = 2*a[1]; // both lvalue and rvalue
```


Operator overloading: increment

```
class Test {
      int i;
public:
      Test(int a = 0) { i = a;}
      Test& operator++() { // ++t, returns ref
            i++;
            return (*this);
      }
      Test operator++(int) { // t++, returns non-ref
            Test tmp = (*this);
            i++;
            return tmp;
      }
};
```


Deep vs shallow copy



Assignment operator

- Default assignment implementation
  - bitwise copy
  - drawback: shallow copy instead of deep
  - programmer is responsible for deep copy
  - take care for memory allocation/deallocation

Assignment operator 2

- Let's copy the *IntArray* class

```
class IntArray {
      ...
  public:
      IntArray& operator=(const IntArray& ia) {
            if (&ia == this) return *this;
            delete [] array;
            size = ia.size;
            array = new int[size];
            for (int i = 0; i < size; i++) {
                  array[i] = ia.array[i];
            }
      }
};
```

allows cascade

Assignment vs. Copy constr.

- Copy constructor initializes a **new** object
  - fields are not yet initialized
  - there is no dynamic field to be deleted
- Assignment modifies the state of an **already existing** object
  - fields already have a value
  - dynamic elements should be deleted before reassigned

Assignment vs. Copy constr. 2

```
int main() {
	IntArray a1, a2;

	a1.add(10);
	a1.add(20);

	a2 = a1; // a2 exists

	IntArray a3 = a1; // a3 is created new
	// IntArray a3(a1);

}
```


Copy constructor in expressions

```
Complex c1(1,2);
Complex c2(2,3);
Complex c3(3,4);

Complex c4 = c1 + c2 + c3;

//Complex tmp1(c1.operator+(c2));
//Complex tmp2(tmp1.operator+(c3));
//tmp1.~Complex();
//Complex c4(tmp2);
//tmp2.~Complex();
```


Constant member functions

- Constant objects are important
  - can be passed as references
  - and won't be changed
  - compiler checks changing
    - it's an error to call a non-const function of a const object
- E.g.

```
int foo(const IntArray& a)
```

Constant member functions 2

- How does compiler check?
  - for primitive types operators
  - for classes const functions

```
class A { ...
    B foo(...) const {...}
    C bar(...) const;
};
```

  - compiler checks constant functions recursively

Constructor initializers

- Let's have the following class

```
class Test {
      int a;
      const int b;
      int& c;
  public:
      Test(int x, int y, int z) {
            a = x; // OK
            b = y; // error
            c = z; // error
      }
};
```

- How do we initialize the reference and constant fields?

Constructor initializers 2

- Initialization before constructor body

```
class Test {
      int a;
      const int b;
      int& c;
  public:
      Test(int x, int y, int z) : b(y), c(z) {
            a = x; // OK
      }
};
```

Reference field

```
class Test {
      int a;
      const int b;
      int& c;
  public:
      Test(int x, int y, int z) : b(y), c(z) {
            a = x; // OK
      }
      void add(int i) { c += i; }
};
```

```
int main() {
      int l = 1; m = 2; n = 3;
      Test t(l,m,n);
      t.add(50);
      cout << n << endl; // 52
}
```


Cast operators

- Cast operators have the following form:
  - `operator TYPE() {...}`
- There is no return value (it's obvious)
- They have to return an expression of the type specified
- A class can have only one cast operator for each type

Cast operators 2

```
class Complex {
      double re;
      double im;
  public:
      ...
      operator double() {
            return sqrt(re*re+im*im);
      }
};
```

```
int main() {
      Complex c(3.0,4.0);
      cout << (double)c; // 5.0, old syntax
      cout << double(c); // 5.0, new syntax
}
```

Implicit conversions

- For primitive types same rules as in C
- For classes cast operators and constructors are also considered
- E.g:

```
class X { // ...
      public : operator int ();
};
class Y { // ...
      public : operator X();
};
Y a;
int b = a;    // error
int c = X(a); // OK: a.operator X().operator int()
```


Implicit conversions 2

- One parameter constructors are called converting constructors

```
class X {
      ...
  public:
      X(int);
      X(const char *, int =0);
};
void f(X arg) {
      X a = 1;    // a = X(1)
      X b = "Jessie";   // b = X("Jessie",0)
      a = 2;      // a = X(2)
      f(3);       // f(X(3))
}
```


Explicit constructor

- *explicit* modifier prohibits implicit conversion

```
class X {
      ...
  public:
      explicit X(int);
};

void f(X arg) {
      X a = 1;  // error
      a = X(2);       // OK
      f(3);     // error
      f(X(3));        // OK
}
```

Conversion example

- What's wrong?

```
class Complex {
    double re, im;
public:
    Complex(double a=0, double b = 0):re(a),im(b) {}
    Complex(Complex& c) : re(c.re),im(c.im) {}
    operator double() const {
       return sqrt(re*re+im*im);
    }
};

int main() {
    const Complex c1(3,4);
    Complex c2 = c1;
}
```


Static fields

- Class-scope fields are called static
- In C++ the modifier *static* is to be used
- Static functions are declared and defined as non-static functions
  - non-static members can not be accessed
- Static variables are defined as globals

Static fields 2

```
class Test {
      int a;
      static int b;
  public:
      static int foo(int x) { return b+x; }
      static int foo2(int x);
      int bar(int y) { return a+b; }
};
int Test::foo2(int x) { return b-x; }
int Test::b = 10; // definition is global
int main() {
      cout << Test::foo(12);  // 22
      cout << Test::foo2(12); // -2
      Test::b = 4; // error: private
}
```

Inheritance, Virtuality, Abstract Classes

Inheritance

- B inherits from A
  - A superclass
  - B subclass
- Inheritance can be
  - analitical (public)
    - B can be used in place of A
  - restrictive (private)
    - B can not be used in place of A
    - limited use

Inheritance example

```
class A {
      int a;
public:
      A(int i) : a(i) {}
      int foo(int k) { return a*k; }
};
class B : public A {
      double d;
  public:
      B(int x, double y) : A(x), d(y) { }
      int bar(int q) { return d*q; }
};

A a(10);
B b(20, 3.4);
```

Visibility

```
class A {
  private: // default
       int a;
  protected:
       int b;
  public:
       int c;
};
class B : public A {
  public:
       int foo() {
             int i = 0;
             i += a; // error, a is private
             i += b; // OK, B is subclass
             i += c; // OK, c is public
       }
};
```


Visibility 2

```
class A {
       void bar() { cout << "A::bar" << endl; }
  public:
       void foo() { cout << "A::foo" << endl; }
       void foo(int i)
             { cout << "A::foo " << i << endl; }
};
class B : public A {
  public:
       void foo(int i)
             { cout << "B::foo " << i << endl; }
};

B b;
b.foo();     // A::foo
b.foo(10);   // B::foo 10
b.bar();     // error, B doesn't have bar() method
```


Constructors

- Code duplication should always be omitted
  - if code is duplicated, it's a design flaw
- Superclass' constructors can be called in the initialization list

```
B(int x, double y) : A(x), d(y) {}
```

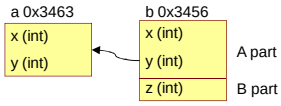
- Copy constructors are to be defined also in initialization list

Constructors example

```
class A {
    int i;
public:
    A(int x) : i(x) { cout << "A::ctr";}
    A(const A& x) : i(x.i) { cout << "A::copy";}
};
class B : public A {
    int j;
public:
    B(int x, int y) : A(x), j(y) { cout << "B::ctr";}
    B(const B& x) : A(x), j(x.j) { cout << "B::copy";}
};
int main() {
       B b1(3,5); // A::ctr, B::ctr
       B b2 = b1; // A::copy, B::copy
}
```

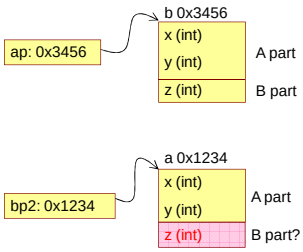

Conversions & memory image

```
class A {
       int x;
       int y;
       ...
};

class B : public A {
       int z;
};

B b;
A a = b;     // OK,trunc.
B b2 = a;    // error
B b3 = (B)a; // error
```



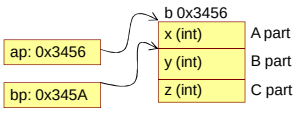
Conversions & memory image

```
class A {
       int x;
       int y;
};

class B : public A {
       int z;
};

B b;
A* ap = &b; // OK
A a;
B* bp = &a; // CT error
B* bp2 =
    (B*)&a; // RT error
```



Conversions & memory image

```
class A {
      int x;
};
class B {
      int y;
};
class C : public A,
      public B {
      int z;
};
C c;
A* ap = &c; //OK,0x3456
B* bp = &c; //OK,0x345A
C* cp = bp; //CT error
C* cp2 =
    (C*)bp; //OK,0x3456
```




```
B b;  // &b = 0x345A
C* cp3 =
    (C*)&b; //RT error,0x3456
```


Casting in C++

- More often than in C
  - in C usually only with *malloc/calloc/realloc*
  - in C++ inheritance, heterogeneous collections
- New functions defined
  - `dynamic_cast<T>(v)` : casts *v* to type *T* (super to sub), checks correctness (NULL)
  - `static_cast<T>(v)` : like dynamic, but compile time, does not check
  - `const_cast<T>(v)` : changes const-ness
  - `reinterpret_cast<T>(v)` : cast with **no** check. DANGEROUS!!!

Virtual functions

- Problem:
  - Let's implement a drawing system.
  - There are shapes that can be lines, circles, rectangles, etc.
  - Let grouping be allowed.

Drawing system design

- Shape
  - position: (x,y)
  - draw( ): empty implementation
  - move (dx, dy):
    - set color background
    - draw( )
    - *x',y' = x+dx,y+dx*
    - set color foreground
    - draw( )

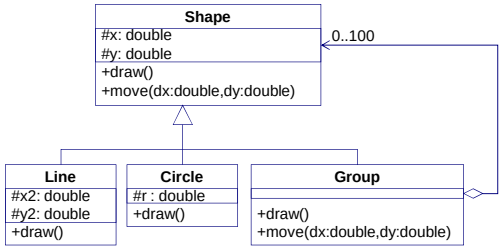
Drawing system design

- Line, subclass of Shape
  - start: position
  - end: (x2,y2)
  - draw( ): draw a line between start and end points
- Circle, subclass of Shape
  - center: position
  - radius: (r)
  - draw( ): draw a circle around center with radius *r*

Drawing system design

- Group, subclass of Shape
  - set of shapes
  - draw: delegate to parts

Drawing system design



Drawing system implementation

```
class Shape {
  protected:
    double x,y;
  public:
    Shape(double a=0, double b=0):x(a),y(b){}
    Shape(const Shape& s):x(s.x),y(s.y){}
    void draw() { cout<<"shape("<<x<<";"<<y<<")"<<endl; }
    void move(double dx, double dy) {
      setColor(BACKGROUND);
      draw();
      x+=dx; y+=dy;
      setColor(FOREGROUND);
      draw();
    }
};
```


Drawing system implementation

```
class Line : public Shape {
  protected:
    double x2,y2;
  public:

    Line(double a=0, double b=0, double c=0, double d=0)
       :Shape(a,b), x2(c),y2(d) {}

    Line(const Line& l) : Shape(l),x2(l.x2),y2(l.y2) {}

    void draw() {
       cout<<"line("<<x<<";"<<y<<" - "
             <<x2<<";"<<y2<<")"<<endl;
    }
};
```

Drawing system implementation

```
class Circle : public Shape {
  protected:
    double r;
  public:

    Circle(double a=0, double b=0, double c=0)
       :Shape(a,b), r(c) {}

    Circle(const Circle& c) : Shape(c),r(c.r) {}

    void draw() {
       cout<<"circle("<<x<<";"<<y<<"; r:"<<r<<")"<<endl;
    }
};
```


Virtual functions

- What is printed?

```
Shape s(1,2);
Circle c(3,4,5);
Line l(1,3,5,7);

s.draw();    //shape(1;2)
c.draw();    //circle(3;4; r=5)
l.draw();    //line(1;3 - 5;7)

s.move(2,2); //shape(1;2) - shape(3;4) OK
c.move(2,2); //shape(3;4) - shape(5;6) ???
l.move(2,2); //shape(1;3) - shape(3;5) ???
```


Virtual functions 2

- What did go wrong?
  - *move* inherited from *Shape* doesn't call *draw* in subclasses
  - *move* calls *this->draw()*
- Solution: virtual functions
  - *this->draw()* should point to subclass' *draw()*
  - what is the type of *this*?
    - **static type**: Shape (compile time)
    - **dynamic type**: Line or Circle (runtime)
  - also holds for references

Virtual functions 3

- By default static type's function are called
- Use *virtual* modifier in order to enable dynamic type's functions to be called:

```
class Shape {
  ...
  public:
    ...
    virtual void draw() {
       cout<<"shape("<<x<<";"<<y<<")"<<endl;
    }
    ...
};
```


Virtual functions 4

- Adding *virtual*, what is printed?

```
Shape s(1,2);
Circle c(3,4,5);
Line l(1,3,5,7);

s.draw();     //shape(1;2)
c.draw();     //circle(3;4; r=5)
l.draw();     //line(1;3 - 5;7)

s.move(2,2); //shape(1;2) - shape(3;4)
c.move(2,2); //circle(3;4; r=5) - circle(5;6; r=5)
l.move(2,2); //line(1;3 - 5;7) - line(3;5 - 7;9)
Shape& s2 = c;
s2.move(2,2); //circle(5;6; r=5) - circle(7;8; r=5)
```


Virtual functions 5

- No pointer, pointer, reference

```
Circle c(1,2,5);
Shape s = c;
Shape& sr = c;
Shape* sp = &c;

c.draw();     //circle(1;2; r=5)

c.move(2,2);   //circle(1;2; r=5) - circle(3;4; r=5)
s.move(2,2);   //shape(1;2) - shape(3;4)
sr.move(2,2);  //circle(3;4; r=5) - circle(5;6; r=5)
sp->move(2,2); //circle(5;6; r=5) - circle(7;8; r=5)
```

Virtual functions 6

- How does it work?
  - for each virtual function there is a pointer in the superclass' memory block
  - it points to the dynamic type's (or the closest implementing superclass') virtual function implementation
  - when called, even if static type is different, dynamic type's function is accessed

Group: heterogeneous collection

```
class Group : public Shape {
  protected:
    Shape parts[100]; // is it OK?
    int size;
  public:
    Group() { size = 0; }
    Group(const Group& g):size(g.size) {
       for (int i = 0; i < size; i++) {
         parts[i] = g.parts[i];
       }
    }
    void add(Shape s) {
       parts[size++] = s; // no size check
    }
    ...
```


Group: heterogeneous collection

```
    ...
    void draw() {
       cout << "Group:"<<endl;
       for (int i = 0; i < size; i++)
         parts[i].draw();
    }
    void move(double dx, double dy) {
       for (int i = 0; i < size; i++)
         parts[i].move(dx,dy);
    }
};
```

Group: heterogeneous collection

- If array is of type ***Shape***
  - adding a new line, circle, etc. truncates
  - we only store the shape part
  - everything will be shape both statically and dynamically
- If array is of type ***Shape****
  - adding a new line, circle, etc doesn't truncate (only pointer conversion)
  - dynamic type is retained, virtual functions work

Group: heterogeneous collection

```
class Group : public Shape {
  protected:
    Shape* parts[100]; // no truncation
    int size;
  public:
    Group() { size = 0; }
    Group(const Group& g):size(g.size) {
       for (int i = 0; i < size; i++) {
         parts[i] = g.parts[i];
       }
    }
    void add(Shape* s) { // must provide ptr
       parts[size++] = s; // no size check
    }
    ...
```


Group: heterogeneous collection

```
    ...
    void draw() {
       cout << "Group:"<<endl;
       for (int i = 0; i < size; i++)
         parts[i]->draw(); // virtual
    }
    void move(double dx, double dy) {
       for (int i = 0; i < size; i++)
         parts[i]->move(dx,dy); // not virt.
    }
};
```

Virtual destructors

```
Group g;
g.add(new Line(3,4,5,6));
g.add(new Circle(1,2,3));
Group* g2 = new Group;
g2->add(new Line(7,6,5,4));
g.add(g2);
g.draw(); // OK, recursive
// what happens when destructed?
// destructor needed
```

```
class Group {
  ...
  ~Group() {
    for (int i=0; i<size; i++) delete parts[i];
  }
};
```


Virtual destructors 2

- What happens when deleting *g2 (g.parts[2])* ?
  - destructor non virtual, so static destructor of Shape is called ☹
  - destructor of Shape has to be virtual

```
class Shape {
  ...
  virtual ~Shape() {}
};
```

- Rule of thumb:

  Superclass always with virtual destructor!

Abstract classes

- Abstract class
  - part of an inheritance hierarchy, but doesn't instantiate
  - usually some method implemented
  - some method left to subclasses
- In C++
  - at least one method defined *pure virtual*
    - virtual modifier and instead of body = 0 defined
    - e.g. `virtual int getSize() = 0;`

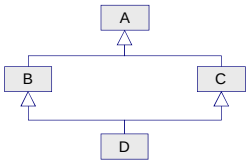
Abstract classes 2

- In previous example *Shape* should be abstract
- It's *draw* method is pointless

```
class Shape {
  ...
  public:
    ...
    virtual void draw() =0;
    ...
};
```

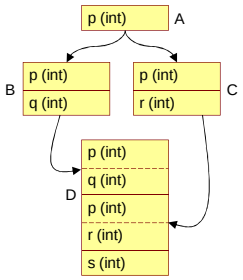

Multiple inheritance revisited

- Multiple inheritance already covered
- What happens in diamond inheritance?



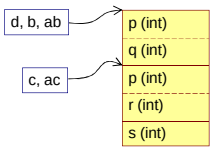
Multiple inheritance revisited 2

```
class A {
        int p;
};
class B:public A {
        int q;
};
class C : public A {
        int r;
};
class D : public B,
    public C {
        int s;
};
```



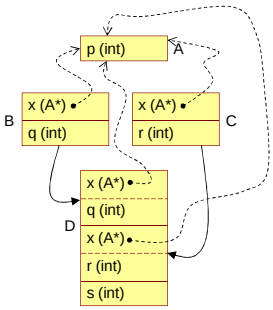
Multiple inheritance revisited 3

```
D* d = new D;

B* b = d;  // OK
C* c = d;  // OK

A* a = d;  // error, ambiguous

A* ab = b; // OK
A* ac = c; // OK

d->q = d->r = d->s = 1; // OK

d->p = 3;  // error, ambiguous
```



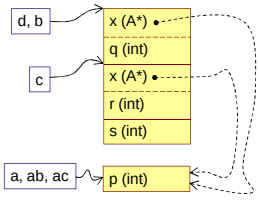
Virtual inheritance

```
class A {
      int p;
};
class B :
virtual public A {
      int q;
};
class C :
virtual public A {
      int r;
};
class D : public B,
    public C {
      int s;
};
```



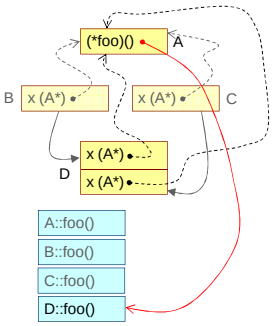
Virtual inheritance 2

```
D* d = new D;

B* b = d;  // OK
C* c = d;  // OK

A* a = d;  // OK, both points
     // to same
A* ab = b; // OK
A* ac = c; // OK

d->q = d->r = d->s = 1; // OK

d->p = 3;  // OK, same field
```



Virtual inheritance 3

```
class A { public:
  virtual void foo();
};
class B : virtual public A {
public:  void foo();
};
class C : virtual public A {
public:  void foo();
};
class D : public B, public C {
public:  void foo();
};
```



Virtual inheritance 4

```
D* d = new D;

B* b = d;  // OK
C* c = d;  // OK

A* a = d;  // OK, both points
     // to same
A* ab = b; // OK
A* ac = c; // OK

a->foo(); // calls D::foo()
b->foo(); // calls D::foo(), even if not virt. inher.
c->foo(); // calls D::foo(), even if not virt. inher.
```


Task of constructor

- Calls constructors virtual superclasses (even if not direct base)
- Calls constructors of direct, non virtual superclasses
- Creates own parts
  - sets virtual superclass pointers
  - sets virtual functions' pointers
  - calls constructor of fields
  - executes constructor body

Task of destructor

- Deletes own part
  - executes destructor body
  - calls destructors of fields
  - resets virtual functions' pointers
  - resets virtual superclass pointers
- Calls destructor of direct, non virtual superclasses
- Calls destructor of virtual superclasses

Virtual inheritance again

```
class A { public:
   int x;
   A(int i) { x = i; }
};
class B : virtual public A {
public: B(int i) : A(i+1){}
};
class C : virtual public A {
public: C(int i) : A(i+2){}
};
class D : public B, public C {
public: D(int i) :
        A(i+3), B(i), C(i){}
};
```

```
A a(1); // x = 1
B b(1); // x = 2
C c(1); // x = 3
D d(1); // x = 4

// if non virtual:
// only B, C ctrs
D d2(1);
// d2.B::x = 2
// d2.C::x = 3
```

mandatory if virtual base
ambiguous if non virtual

Namespaces, Exceptions, etc.

Namespace

- Names are scarce
  - consider *Test, foo, address, time, length, complex,* etc.
  - when mixing modules name collisions are unavoidable
  - in C++ names can be put into separate namespaces
- *namespace* keyword
  - declares a namespace
- Standard libraries use namespace *std*
  - hence `using namespace std;`

Namespace example

```
namespace example {
	const double pi = 3.1415;
	int counter = 0;
	class Test {};
}
namespace test {
	int pi = 3; // approximation
	class Test {};
}

pi++; // error, not defined
test::pi++; // OK, namespace defined
class Test2 : public test::Test {};
::std::cout << "Hello world" << ::std::endl;
```


Using namespaces

- Multiple declaration is allowed
  - works like 'opening' namespace again
- Namespaces can be nested
  - `test1::test2::test3`
- Default namespace (root) exists: `::`
  - `::test1::test2::test3`
- Namespaces or parts can be imported
  - `using ::example::pi;`
  - `using namespace std;`

Namespace example 2

```
namespace example {
	const double pi = 3.1415;
	int counter = 0;
	class Test {};
}
namespace test {
	int pi = 3; // approximation
	class Tester {};
}

using example::Test
class Test2 : public Test {}; // OK, Test is imported
double r = 4.5;
double area = r*r*pi; // error, pi not imported
using namespace test; // imports whole namespace
```


Error handling

- In C errors are reported through return values:
  - `FILE* f = fopen("/home/joe/text.txt", "r");`
  - `int c = getchar();`
  - `errno` may bet set accordingly
- Problem:
  - return value and error notifications are mixed
  - error handling code is also mixed

Error handling 2

```
int main(int argc, char** argv) { // C error handling
	int a,b; int err;
	File* f = new fopen("/home/joe/text.txt", "r");
	if (f == NULL) {
		fprintf(stderr, "%s: %s",
			argv[0], strerror(errno));
		exit(-1);
	}
	err = fscanf(f, "%d %d", &a, &b);
	if (err < 2) {
		fprintf(stderr, "%s: Wrong format!!!",
			argv[0]);
		exit(-2);
	}
	printf("%d\n", a*b);
}
```

C++ error handling: exceptions

- Exceptions are notifiers of problems
- Their type can be primitive or compound
- Error detection and error handling is separated
  - *throw* for notification
  - *try* block for detection
  - *catch* block for handling
- Clean design
  - no base-meta mixup

Exception example: IntArray

```
class IntArray {
      ...
public:
      ...
      class BoundsException {} // inner class
      int& operator[](int index) {
            if (index < 0 || index >= size) {
                  throw BoundsException();
            }
            return array[index];
      }
};
```


Exception example: IntArray

```
int main() {
      IntArray ia;
      ia.add(10);
      ia.add(4);
      ia.add(7);
      try {
            ia[1] = 2
            ia[10] = 3; // here exc. thrown
            ia[0] = 1;  // not accessed
            ia[-1] = 5; // exc. would be thrown
      } catch (IntArray::BoundsException& e) {
            cerr << "bounds exception" << endl;
      }
      for (int i = 0; i< ia.getSize(); i++)
            cout << ia[i] << endl;
      // 10, 2, 7
}
```

Exception handling

- Any number of catch blocks
- Exception – catch matching by type, works like function call
  - inheritance can be used
  - always most specific first
  - it's advised to use exception classes and inheriting specific exceptions
  - it's advised to use reference or pointer types

Exception handling 2

- `throw` without parameter throws the caught exception → original type is used
- `catch(...)` catches all kinds of exceptions (even primitive types)
- dynamic objects might not be deleted
  - use `std::auto_ptr`
  - works like smart pointer
  - automatically deletes pointer if deleted

Exception and pointers

```
try {
       A a; // deleted even after throw
       B* bp = new B;
       throw Exception();
       delete bp; // might not be called
} catch (Exception& e) {
}
```

```
try {
       A a;
       std::auto_ptr<B> bp(new B);
       throw Exception();
       // bp automatically deleted
} catch (Exception& e) {
}
```

Exception and (con|de)structors

```
class A {
public: A() { throws Exception(); }
        ~A() { throws Exception(); }
};
class B : public A {
public:
        B() : A() {
                try { ... }
                catch (...) {} // is it enough?
        }
        ~B() {
                try { ... }
                catch (...) {} // is it enough?
        }
};
```


Exception and (con|de)structors

```
class A {
public: A() { throws Exception(); }
        ~A() { throws Exception(); }
};
class B : public A {
public:
        B() : A() try {
                ...
        } catch (...) {...} // catches outside

        ~B() try {
                ...
        } catch (...) {...}

};
```


Specifying exceptions

- It is good practice to specify exceptions thrown by a method
  - `void foo() throw(E1, E2);`
    - only E1 and E2
  - `void foo() throw();`
    - nothing
  - `void foo();`
    - anything
  - every other exception makes `unexpected()` called

Unexpected exception

- Unexpected exceptions make `unexpected()` to be called
  - it calls function pointed to be `unexpected_handler`
  - default call: `terminate()`
- Modifying handler:
  - `set_unexpected(void (*handler)())`
- The provided function may throw new exceptions

Templates

Dynamic array again

- Problem:

  Let's modify *IntArray* to hold *double* values

IntArray *int* specifics

```
class IntArray {
  int* array; // the integers
  int size;   // size of the array
 public:
  IntArray() { size = 0; array = new int[0]; }
  IntArray(const IntArray& ia) {
    size = ia.size;
    array = new int[size];
    for (int i = 0; i < size; i++) {
      array[i] = ia.array[i];
    }
  }
  ~IntArray() { delete [] array; }
  ...
```


IntArray *int* specifics 2

```
  ...
  void add(int d) {
    int* tmp = new int[size+1];
    for (int i = 0; i < size; i++) {
      tmp[i] = array[i];
    }
    tmp[size] = d;
    delete [] array;
    array = tmp;
    size++;
  }
  int& get(int index) { return array[index]; }
  int& operator[](int i) { return array[i]; }
  int getSize() { return size; }
};
```


DoubleArray?

- Only parts marked red should be changed to *double*
- Automatic change (search/replace)
  - would change other int-s also
    - e.g. `int getSize() { return size; }`
- Better idea: have specific type T, and change T to type needed
  - this is what templates are for

Template array

```
template <class T> class Array {
  T* array; // the integers
  int size;   // size of the array
 public:
  Array() { size = 0; array = new T[0]; }
  Array(const Array& ia) {
    size = ia.size;
    array = new T[size];
    for (int i = 0; i < size; i++) {
      array[i] = ia.array[i];
    }
  }
  ~Array() { delete [] array; }
  ...
```


Template array 2

```
  ...
  void add(T d) {
    T* tmp = new T[size+1];
    for (int i = 0; i < size; i++) {
      tmp[i] = array[i];
    }
    tmp[size] = d;
    delete [] array;
    array = tmp;
    size++;
  }
  T& get(int index) { return array[index]; }
  T& operator[](int i) { return array[i]; }
  int getSize() { return size; }
};
```


Template array 3

```
int main() {

      Array<int> ia;
      Array<double> da;
      ia.add(1);
      da.add(2.3);
      ia.add(2);
      da.add(4.8);
      da[0] = 7.2;

}
```

Template function definition

```
// overindexing allowed: dynamic growth
template <class T> T& Array<T>::operator[](int i) {
	if (i < 0) throw BoundsException();
	if (i >= size) {
		T* tmp = new T[i+1];
		for (int j = 0; j < size; j++)
			tmp[j] = array[j];
		for (int j=size; j < i+1; j++)
			tmp[i] = T(); //default ctr
		delete [] array;
		array = tmp;
		size = i+1;
	}
	return array[i];
}
```


Template rules

- Any number of template parameters
  - `template <class T, int s, class U>`
- Default constructor also for primitive types
  - initializes to 0
- Template keyword lasts till the end of marked declaration/definition
- For each template instance new code is generated

Template rules 2

- Always mention prerequisites of template parameters
  - default constructor
  - copy constructor
  - assignment operator
  - less than, etc operators
  - etc.

Template function: maximum

- #define vs. inline
  - parameters: inline correct
  - speed: no difference
  - types: define for all, inline different for each
- using templates:

```
template <class T> inline T max(const T& a, const T& b) {
      return (a>b)?a:b;
}

int a = max(10,20); // <int> known from parameters
```


Template specialization

- What happens:

```
template <class T>
inline T max(const T& a, const T& b) {
      return (a>b)?a:b;
}

cout << max(10,20) << endl;    // 20
cout << max(1.3, 4.5) << endl; // 4.5
const char* s1 = "apple", *s2 = "pear";
cout << max(s1, s2) << endl; ????
```


Template specialization 2

- We have to specialize for *char**

```
template <class T>
inline T max(const T& a, const T& b) {
      return (a>b)?a:b;
}
const char* max(const char* a, const char* b) {
      return (strcmp(a,b)>0)?a:b;
}

cout << max(10,20) << endl;    // 20
const char* s1 = "apple", *s2 = "pear";
cout << max(s1, s2) << endl; // now OK: pear
```

Quicksort in C++

```
template <class T> class Sorter {
  static int cmp(T a, T b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
  }
  static int cmp0(const void* a, const void* b) {
    T* ta = (T*)a;
    T* tb = (T*)b;
    return cmp(*ta, *tb);
  }
public:
  static void sort(T* t, int n) {
       qsort(t, n, sizeof(T), cmp0);
    }
};
```


Quicksort in C++ 2

```
int main() {
    int* array = new int[5];
    array[0]=3;
    array[1]=1;
    array[2]=2;
    array[3]=5;
    array[4]=4;

    Sorter<int>::sort(array,5);

    for (int i = 0; i < 5; i++) {
        cout << array[i] << endl;
    }
}
```


Quicksort with pointers

- What can we do, if the parameter is *char**
  - specialization!

```
template <>
int Sorter<char*>::cmp(char* a, char *b) {
       return strcmp(a, b);
}
...
char** str = new char*[3];
str[0] = "c";
str[1] = "a";
str[2] = "b";
Sorter<char*>::sort(str, 3);
```

Quicksort for classes

- Let's have the following class
  - how do we specialize?

```
class Student {
      char* name;
      int year_of_birth;
      double average;
      ...
};
```

  - `int operator<(const Student& s)` ???
  - specialization like *char** ???
  - problem: more than one way of sorting

Quicksort with predicate

```
template <class T> class Comparator {
 public:
  static int cmp(T a, T b) {
    if (a < b ) return -1;
    if (b < a ) return 1; else return 0;
  }
};
template <class T, class C = Comparator<T> >
class Sorter {
  static int cmp(const void* a, const void*b) {
    return C::cmp(*(T*)a, *(T*)b);
  }
 public:
  static void sort(T* t, int n)
    { qsort(t, n, sizeof(T), cmp); }
};
```

predicate with default value

Quicksort with predicate 2

```
// for char* function specialization
template <>
int Comparator<char*>::cmp(char* a,
                           char* b) {
  return strcmp(a, b);
}

// for student sort by average with new class
class StudentAverageCmp {
 public:
  static int cmp(Student& a, Student& b) {
    return Comparator<double>
           ::cmp(a.getAverage(), b.getAverage());
  }
};
```

Quicksort with predicate 3

```
int main() {
  Student* st = new Student[3];
  st[0] = Student("Adam", 1985, 4.3);
  st[1] = Student("Joe", 1983, 3.6);
  st[2] = Student("Zeno", 1986, 2.1);

  Sorter<Student, StudentAverageCmp>::sort(st, 3);

  for (int i = 0; i < 3; i++) {
       cout << st[i]; << endl;
  }
}
```


Template rules of thumb

- The interface of parameter types is not explicit
  - always document parameter type responsibilities
- Use templates with care
  - for each parameter type a new class/function is generated

Standard Template Library

- Library with lots of helpful classes, templates, algorithms, functions, etc.
- For example:
  - string class, comparison, concatenation, ...
  - sort, swap, fill, for_each, search, count_if, ...
  - vector, hashmap, iterator, ...
- http://www.cplusplus.com/reference/stl/

Bibliography

- Bjarne Stroustrup: *The C++ Programming Language.* Addison-Wesley Professional, 1997
- `www.cplusplus.com`