Object Oriented Software Design II

Introduction to C++

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

February 20, 2012

Outline

Outline

Prerequisites

- To understand this course, you should at least know the basic C syntax
  - functions declaration and function call,
  - global and local variables
  - pointers (will do again during the course)
  - structures
- First part of the course: classes
  - Classes, objects, memory layout
  - Pointer and references
  - Copying
  - Inheritance, multiple inheritance
  - Access rules
  - Public, protected and private inheritance
  - Exceptions

Summary - cont.

- Second part: templates
  - Templates
  - The Standard Template Library
- Third part: new standard
  - What does it change
  - lambda functions
  - auto
  - move semantic
  - new STL classes
  - Safety to exceptions
- Fourth part: patterns
  - Some patterns in C++
  - Function objects
  - Template patterns
  - Meta-programming with templates
- Fifth part: libraries
  - Thread library, synchronization
  - Futures and promises
  - The Active Object pattern

Outline

Classes

C is not a high-level language.

– Brian Kernighan, inventor of C with D. M. Ritchie

Those types are not abstract: they are as real as int and float

– Doug McIlroy

Actually I made up the term 'object-oriented', and I can tell you I did not have C++ in mind.

– Alan Kay

Abstraction

- An essential instrument for OO programming is the support for data abstraction
- C++ permits to define new types and their operations
- Creating a new data type means defining:
  - Which elements it is composed of (*internal structure*);
  - How it is built/destroyed (*constructor/destructor*);
  - How we can operate on this type (*methods/operations*).

Data abstraction in C

- We can do data abstraction in C (and in almost any language)

```
typedef struct __complex {
    double real_;
    double imaginary_;
} cmplx;

void add_to(cmplx *a, cmplx *b);
void sub_from(cmplx *a, cmplx *b);
double get_module(cmplx *a);
```

- We have to pass the main data to every function
- name clashing: if another abstract type defines a function `add_to()`, the names will clash!
- No information hiding: any user can access the internal data using them improperly

Classical example

```
class Complex {
   double real_;
   double imaginary_;
public:
    Complex();
    Complex(double a, double b);
    ~Complex();

    double real() const;
    double imaginary() const;
    double module() const;
    Complex &operator =(const Complex &a);
    Complex &operator+=(const Complex &a);
    Complex &operator-=(const Complex &a));
};
```

How to use complex

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

cout << "c1=(" << c1.real() << ","
     << c1.imaginary() << ")" << endl;

c1  = c2;            // assignment
c3 += c1;            // operator +=
c1  = c2 + c3;       // ERROR: operator + not yet defined
```

Using new data types

- The new data type is used just like a predefined data type
  - it is possible to define new functions for that type:
    - `real()`, `imaginary()` and `module()`
  - It is possible to define new operators
    - =, += and `-=`
  - The compiler knows automatically which function/operator must be invoked
- C++ is a strongly typed language
  - the compiler knows which function to invoke by looking at the type

Outline

Class

- Class is the main construct for building new types in C++
  - A class is almost equivalent to a struct with functions inside
  - In the C-style programming, the programmer defines structs, and global functions to act on the structs
  - In C++-style programming, the programmer defines classes with embedded functions

```
class MyClass {
  int a;
public:
  int myfunction(int param);
};
```

Class declaration

Remember the semicolon!

Members

- A class contains members
- A member can be
  - any kind of variable (member variables)
  - any kind of function (member functions or methods)

```
class MyClass {
  int a;
  double b;
public:
   int c;

   void f();
   int getA();
   int modify(double b);
};
```

Members

- A class contains members
- A member can be
  - any kind of variable (member variables)
  - any kind of function (member functions or methods)

```
class MyClass {
  int a;
  double b;
public:
   int c;

   void f();
   int getA();
   int modify(double b);
};
```

member variables (private)

Members

- A class contains members
- A member can be
  - any kind of variable (member variables)
  - any kind of function (member functions or methods)

```
class MyClass {
  int a;
  double b;
public:
   int c;

   void f();
   int getA();
   int modify(double b);
};
```

member variables (private)

member variable (public)

Members

- A class contains members
- A member can be
  - any kind of variable (member variables)
  - any kind of function (member functions or methods)

```
class MyClass {
  int a;
  double b;
public:
   int c;

   void f();
   int getA();
   int modify(double b);
};
```

member variables (private)

member variable (public)

member functions (public)

Declaring objects of a class: constructor

- An **object** is an instance of a class
- An object is created by calling a special function called *constructor*
  - A constructor is a function that has the same name of the class and no return value
  - It may or may not have parameters;
  - It is invoked in a special way

```
class MyClass {
public:
  MyClass()
  {
    cout << "Constructor"<<endl;
  }
};

MyClass obj;
```

Declaration of the constructor

Invoke the constructor to create an object

Constructor - II

- Constructors with parameters

```
class MyClass {
  int a;
  int b;
public:
  MyClass(int x);
  MyClass(int x, int y);
};

MyClass obj;
MyClass obj1(2);
MyClass obj2(2,3);

int myvar(2);
double pi(3.14);
```

A class can have many constructors

- `MyClass obj;` — This is an error, no constructor without parameters
- `MyClass obj1(2);` — Calls the first constructor
- `MyClass obj2(2,3);` — Calls the second constructor
- `int myvar(2);` — Same syntax is valid for primitive data types

Default constructor

- Rules for constructors
  - If you do not specify a constructor, a default one with no parameters is provided by the compiler
  - If you provide a constructor (any constructor) the compiler will not provide a default one for you
- Constructors are used to initialise members

```
class MyClass {
  int a;
  int b;
public:
  MyClass(int x, int y)
  {
    a = x; b = 2*y;
  }
};
```

Initialization list

- Members can be initialised through a special syntax
  - This syntax is preferable (the compiler can catch some obvious mistake)
  - use it whenever you can (i.e. almost always)

```
class MyClass {
  int a;
  int b;
public:
  MyClass(int x, int y) :
    a(x), b(y)
  {
     // other initialisation
  }
};
```

A comma separated list of constructors, following the :

Accessing member objects

- Members of one object can be accessed using the *dot* notation, similarly to structs in C

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x

Accessing member objects

- Members of one object can be accessed using the *dot* notation, similarly to structs in C

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Accessing member objects

- Members of one object can be accessed using the *dot* notation, similarly to structs in C

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x (x.a = 5;)

Assigning to a member variable of object y (y.a = 7;)

Calling member function f() of object x (x.f();)

Accessing member objects

- Members of one object can be accessed using the *dot* notation, similarly to structs in C

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

- `x.a = 5;` — Assigning to a member variable of object x
- `y.a = 7;` — Assigning to a member variable of object y
- `x.f();` — Calling member function f() of object x
- `y.g(5, 10);` — Calling member function g() of object y

Implementing member functions

- You can implement a member function (including constructors) in a separate .cpp file

complex.h

```
class Complex {
  double real_;
  double img_;
public:
  ...
  double module() const;
  ...
};
```

complex.cpp

```
double Complex::module()
{
  double temp;
  temp = real_ * real_ +
         img_ * img_;
  return temp;
}
```

- This is preferable most of the times
- put implementation in include files only if you hope to use *in-lining* optimisation

Accessing internal members

```
double Complex::module() const
{
  double temp;
  temp = real_ * real_ + img_ * img_;
  return temp;
}
```

Accessing internal members

```
double Complex::module() const
{
  double temp;
  temp = real_ * real_ + img_ * img_;
  return temp;
}
```

scope resolution

- The `::` operator is called *scope resolution operator*

Accessing internal members

```
double Complex::module() const
{
  double temp;
  temp = real_ * real_ + img_ * img_;
  return temp;
}
```

scope resolution

- The `::` operator is called *scope resolution operator*
- like any other function, we can create local variables

Accessing internal members

```
double Complex::module() const
{
  double temp;
  temp = real_ * real_ + img_ * img_;
  return temp;
}
```

scope resolution

access to internal variable

- The `::` operator is called *scope resolution operator*
- like any other function, we can create local variables
- member variables and functions can be accessed without *dot* or *arrow*

Outline

Access control

- A member can be:
  - private: only member functions of the same class can access it; other classes or global functions can't
  - protected: only member functions of the same class or of derived classes can access it: other classes or global functions can't
  - public: every function can access it

```
class MyClass {
private:
    int a;
public:
     int c;
};
```

```
MyClass data;

cout << data.a;   // ERROR!
cout << data.c;   // OK
```

Access control

- Default is private
- An access control keyword defines access until the next access control keyword

```
class MyClass {
    int a;
    double b;
public:
     int c;

     void f();
     int getA();
private:
     int modify(double b);
};
```

- private (default)
- public
- private again

Access control and scope

```cpp
int xx;

class A {
   int xx;
public:
   void f();
};
```

global variable

member variable

```cpp
void A::f()
{
   xx = 5;
   ::xx = 3;

   xx = ::xx + 2;
}
```

access local xx

access global xx

Why access control?

- The technique of declaring private members is also called encapsulation
    - In this way we can precisely define what is interface and what is implementation
    - The public part is the interface to the external world
    - The private part is the implementation of that interface
    - When working in a team, each group take care of a module
    - To ensure that the integration is done correctly and without problems, the programmers agree on interfaces

Private

- Some people think that private is synonym of secret
  - they complain that the private part is visible in the header file
- private means not accessible from other classes and does not mean secret
- The compiler needs to know the size of the object, in order to allocate memory to it
  - In an hypothetical C++, if we hide the private part, the compiler cannot know the size of the object

Friends

```
class A {
   friend class B;
   int y;
   void f();
public:
   int g();
};

class B {
   int x;
public:
   void f(A &a);
};

void B::f(A &a)
{
    x = a.y;
    a.f();
}
```

B is friend of A

B can access private members of A

Friend functions and operator

- Even a global function or a single member function can be friend of a class

```
class A {
    friend B::f();
    friend h();
    int y;
    void f();
public:
    int g();
};
```

friend member function → `friend B::f();`

friend global function → `friend h();`

- It is better to use the *friend* keyword only when it is really necessary because it breaks the access rules.
- *"Friends, much as in real life, are often more trouble than their worth." – Scott Meyers*

Nested classes

- It is possible to declare a class inside another class
- Access control keywords apply

```
class A {
    class B {
        int a;
    public:
        int b;
    }
    B obj;
public:
    void f();
};
```

- Class B is private to class A: it is not part of the interface of A, but only of its implementation.
- However, A is not allowed to access the private part of B!! (`A::f()` cannot access `B::a`).
- To accomplish this, we have to declare A as friend of B

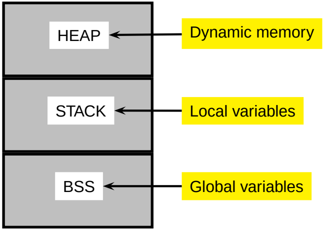
Outline

Memory layout

- Let us recapitulate the rules for the lifetime and visibility of variables
  - **Global variables** are defined outside of any function. Their lifetime is the duration of the program: they are created when the program is loaded in memory, and deleted when the program exits
  - **Local variables** are defined inside functions or inside code blocks (delimited by curly braces { and }). Their lifetime is the execution of the block: they are created before the block starts executing, and destroyed when the block completes execution
- Global and local variables are in different **memory segments**, and are managed in different ways

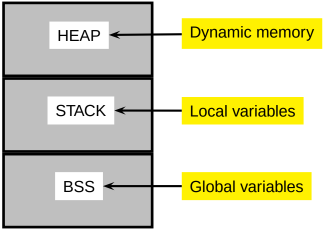
Memory segments

- The main data segments of a program are shown below
- The BSS segment contains **global variables**. It is divided into two segments, one for initialised data (i.e. data that is initialised when declared), and non-initialised data.
  - The size of this segment is statically decided when the program is loaded in memory, and can never change during execution



Memory segments

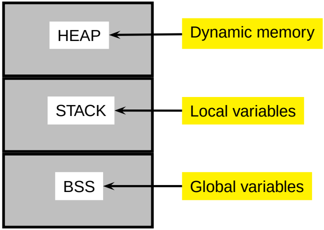
- The main data segments of a program are shown below
- The STACK segment contains **local variables**
  - Its size is dynamic: it can grow or shrink, depending on how many local variables are in the current block



Memory segments

- The main data segments of a program are shown below

- The HEAP segment contains **dynamic memory** that is managed directly by the programmer



Example

- Here is an example:

```
int a = 5; // initialised global data
int b;     // non initialised global data

int f(int i)    // i, d and s[] are local variables
{               // will be created on the stack when
  double d;     // function f() is invoked
  char s[] = "Lipari";
  ...
}

int main()
{
  int s, z;     // local variables, are created on the stack
                // when the program starts

  f();          // here f() is invoked, so the stack for f() is created
}
```

Outline

Pointer

- A pointer is a variable that can hold a memory address
- Basic syntax:

```
int a = 5;
int b = 7;
int *p;

p = &a;

cout << p << endl;

cout << *p << endl;

*p = 6;

p = &b;

cout << *p << endl;
```

Declaration of a pointer to an integer variable

Pointer

- A pointer is a variable that can hold a memory address
- Basic syntax:

```
int a = 5;
int b = 7;
int *p;

p = &a;

cout << p << endl;

cout << *p << endl;

*p = 6;

p = &b;

cout << *p << endl;
```

Declaration of a pointer to an integer variable

p takes the address of a

Pointer

- A pointer is a variable that can hold a memory address
- Basic syntax:

```
int a = 5;
int b = 7;
int *p;

p = &a;

cout << p << endl;

cout << *p << endl;

*p = 6;

p = &b;

cout << *p << endl;
```

Declaration of a pointer to an integer variable

p takes the address of a

print the address

Pointer

- A pointer is a variable that can hold a memory address
- Basic syntax:

```
int a = 5;
int b = 7;
int *p;

p = &a;

cout << p << endl;

cout << *p << endl;

*p = 6;

p = &b;

cout << *p << endl;
```

- `int *p;` — Declaration of a pointer to an integer variable
- `p = &a;` — p takes the address of a
- `cout << p << endl;` — print the address
- `cout << *p << endl;` — prints the value in a

Pointer

- A pointer is a variable that can hold a memory address
- Basic syntax:

```
int a = 5;
int b = 7;
int *p;

p = &a;

cout << p << endl;

cout << *p << endl;

*p = 6;

p = &b;

cout << *p << endl;
```

Declaration of a pointer to an integer variable

p takes the address of a

print the address

prints the value in a

changes the value in a = 6

Pointer

- A pointer is a variable that can hold a memory address
- Basic syntax:

```
int a = 5;
int b = 7;
int *p;

p = &a;

cout << p << endl;

cout << *p << endl;

*p = 6;

p = &b;

cout << *p << endl;
```

- `int *p;` — Declaration of a pointer to an integer variable
- `p = &a;` — p takes the address of a
- `cout << p << endl;` — print the address
- `cout << *p << endl;` — prints the value in a
- `*p = 6;` — changes the value in a = 6
- `p = &b;` — now p points to b

Pointer

- A pointer is a variable that can hold a memory address
- Basic syntax:

```
int a = 5;
int b = 7;
int *p;

p = &a;

cout << p << endl;

cout << *p << endl;

*p = 6;

p = &b;

cout << *p << endl;
```

- `int *p;` — Declaration of a pointer to an integer variable
- `p = &a;` — p takes the address of a
- `cout << p << endl;` — print the address
- `cout << *p << endl;` — prints the value in a
- `*p = 6;` — changes the value in a = 6
- `p = &b;` — now p points to b
- `cout << *p << endl;` — prints the value in b

Arrays

- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
cout << endl;
```

prints “G”

Arrays

- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
cout << endl;
```

prints “G”

declares a pointer to the first element of the array

Arrays

- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
cout << endl;
```

- `cout << *name << endl;` — prints “G”
- `char *p = name;` — declares a pointer to the first element of the array
- `p++;` — Increments the pointer, now points to “i”

Arrays

- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
cout << endl;
```

- `cout << *name << endl;` — prints “G”
- `char *p = name;` — declares a pointer to the first element of the array
- `p++;` — Increments the pointer, now points to “i”
- `assert(p == name+1);` — this assertion is correct

Arrays

- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
cout << endl;
```

- `cout << *name << endl;` — prints “G”
- `char *p = name;` — declares a pointer to the first element of the array
- `p++;` — Increments the pointer, now points to “i”
- `assert(p == name+1);` — this assertion is correct
- `while (*p != 0)` — zero marks the end of the string

Arrays

- The name of an array is equivalent to a constant pointer to the first element
- With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";

cout << *name << endl;

char *p = name;

p++;

assert(p == name+1);

while (*p != 0)
    cout << *(p++);
cout << endl;
```

- `cout << *name << endl;` — prints “G”
- `char *p = name;` — declares a pointer to the first element of the array
- `p++;` — Increments the pointer, now points to “i”
- `assert(p == name+1);` — this assertion is correct
- `while (*p != 0)` — zero marks the end of the string
- `cout << *(p++);` — prints the content of address pointed by p, and increments it

Dynamic memory

- Dynamic memory is managed by the user
- In C:
  - to allocate memory, call function `malloc`
  - to deallocate, call `free`
  - Both take pointers to any type, so they are not type-safe
- In C++
  - to allocate memory, use operator `new`
  - to deallocate, use operator `delete`
  - they are more type-safe

The new operator

- The `new` and `delete` operators can be applied to primitive types, and classes
- `operator new` automatically calculates the size of memory to be allocated

```
int *p = new int(5);

class A { ... };

A *q = new A();

delete p;

delete q;
```

Allocates an integer pointed by p

The new operator

- The `new` and `delete` operators can be applied to primitive types, and classes
- `operator new` automatically calculates the size of memory to be allocated

```
int *p = new int(5);

class A { ... };

A *q = new A();

delete p;

delete q;
```

Allocates an integer pointed by p

Does two things:
1) Allocates memory for an object of class A
2) calls the constructor of A()

The new operator

- The `new` and `delete` operators can be applied to primitive types, and classes
- `operator new` automatically calculates the size of memory to be allocated

```
int *p = new int(5);

class A { ... };

A *q = new A();

delete p;

delete q;
```

Allocates an integer pointed by p

Does two things:
1) Allocates memory for an object of class A
2) calls the constructor of A()

Deallocates the memory pointed by p

The new operator

- The `new` and `delete` operators can be applied to primitive types, and classes
- `operator new` automatically calculates the size of memory to be allocated

```
int *p = new int(5);

class A { ... };

A *q = new A();

delete p;

delete q;
```

Allocates an integer pointed by p

Does two things:
1) Allocates memory for an object of class A
2) calls the constructor of A()

Deallocates the memory pointed by p

Does two things:
1) Calls the *destructor* for A
2) deallocates the memory pointed by q

Destructor

- The destructor is called just before the object is deallocated.
- It is always called both for all objects (allocated on the stack, in global memory, or dynamically)
- If the programmer does not define a constructor, the compiler automatically adds one by default (which does nothing)
- Syntax

```
class A {
    ...
public:
    A() { ... }  // constructor
    ~A() { ... } // destructor
};
```

The destructor never takes any parameter

Example

See `./examples/01.summary-examples/destructor.cpp`

Why a destructor

- A destructor is useful when an object allocates memory
- so that it can deallocate it when the object is deleted

```
class A { ... };

class B {
    A *p;
public:
    B() {
        p = new A();
    }
    ~B() {
        delete p;
    }
};
```

- p is initialised when the object is created
- The memory is deallocated when the object is deleted

New and delete for arrays

- To allocate an array, use this form

```
int *p = new int[5]; // allocates an array of 5 int
...
delete [] p;         // notice the delete syntax

A *q = new A[10];    // allocates an array of 10
...                  // objects of type A
delete [] q;
```

- In the second case, the default constructor is called to build the 10 objects
- Therefore, this can only be done is a default constructor (without arguments) is available

Null pointer

- The address 0 is an invalid address
  - (no data and no function can be located at 0)
- therefore, in C/C++ a pointer to 0 is said to be a *null pointer*, which means a pointer that points to nothing.
- Dereferencing a null pointer is always a bad error (null pointer exception, or segmentation fault)
- In C, the macro NULL is used to mark 0, or a pointer to 0
  - however, 0 can be seen to be of integer type, or a null pointer
- In the new C++, the null pointer is indicated with the constant `nullptr`
  - this constant cannot be automatically converted to an integer

Outline

Function overloading

- In C++, the argument list is part of the name of the function
  - this mysterious sentence means that two functions with the same name but with different argument list are considered two different functions and not a mistake
- If you look at the internal name used by the compiler for a function, you will see three parts:
  - the class name
  - the function name
  - the argument list

Function overloading

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
__A_f_int
__A_f_int_int
__A_f_double
__B_f_int
```

- To the compiler, they are all different functions!
- beware of the type...

Which one is called?

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

Which one is called?

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

__A_f_int

Which one is called?

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

__A_f_int

__B_f_int

Which one is called?

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

- a.f(5); → `__A_f_int`
- b.f(2); → `__B_f_int`
- a.f(3.0); → `__A_f_double`

Which one is called?

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

__A_f_int

__B_f_int

__A_f_double

__A_f_int_int

Which one is called?

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

```
A a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```

- a.f(5); ← __A_f_int
- b.f(2); ← __B_f_int
- a.f(3.0); ← __A_f_double
- a.f(2,3); ← __A_f_int_int
- a.f(2.5, 3); ← __A_f_int_int

Return values

- Notice that return values are not part of the name
  - the compiler is not able to distinguish two functions that differs only on return values

```
class A {
    int floor(double a);
    double floor(double a);
};
```

- This causes a compilation error
- It is not possible to overload a return value

Default arguments in functions

- Sometime, functions have long argument lists
- Some of these arguments do not change often
  - We would like to set default values for some argument
  - This is a little different from overloading, since it is the same function we are calling!

```
int f(int a, int b = 0);

f(12);    // it is equivalent to f(12,0);
```

- The combination of overloading with default arguments can be confusing
- it is a good idea to avoid overusing both of them

Outline

Time to do an example

- Let us implement a Stack of integers class

```
Stack stack;
...
stack.push(12);
stack.push(7);
...
cout << stack.pop();
cout << stack.pop();
```

Interface

```
class Stack {
   ...
public:
    Stack(int maxsize);
    ~Stack();

    void push(int a);
    int pop();
    int peek();
    int size();
};
```

Constructor: maxsize is the maximum number of elements on the stack

- Hint: Use an array to store the elements

Interface

```
class Stack {
   ...
public:
    Stack(int maxsize);
    ~Stack();

    void push(int a);
    int pop();
    int peek();
    int size();
};
```

Constructor: maxsize is the maximum number of elements on the stack

Destructor

- Hint: Use an array to store the elements

Interface

```
class Stack {
   ...
public:
    Stack(int maxsize);
    ~Stack();

    void push(int a);
    int pop();
    int peek();
    int size();
};
```

Constructor: maxsize is the maximum number of elements on the stack

Destructor

Returns the top element

- Hint: Use an array to store the elements

Interface

```
class Stack {
   ...
public:
    Stack(int maxsize);
    ~Stack();

    void push(int a);
    int pop();
    int peek();
    int size();
};
```

Constructor: maxsize is the maximum number of elements on the stack

Destructor

Returns the top element

Returns the current number of elements

- Hint: Use an array to store the elements