OntarioTech Science

Kourosh Davoudi
kourosh@uoit.ca

Week 12: Review

CSCI 1061: Programming Workshop II

Learning Outcomes

In this week, we review the course:

- Functions
- Command Line Arguments
- Pointers
- Encapsulation
- Operator Overloading
- Inheritance
- Polymorphism
- Template Function/Class
- Containers in STL
- Generic Function in STL
- Exceptions

Functions

Function Basic Concepts

- Function Declaration (Prototype)
- Default parameters
- Call by reference vs. Call by value
- Function overloading

Example

```
double volume(int, int = 1,int = 1); // function prototype with default parameters
void swap(int &,int &);

void print_area(double);             // function overloading
void print_area(double,double);

int main(int argc, char const *argv[])
{
    cout << volume(2) << endl;  // function call
    cout << volume(2,3) << endl;
    cout << volume(2,3,4) << endl;

    print_area(3.0);
    print_area(2.0,3.0);

    int x = 1 , y =2 ;

    cout << "before: " << x << " " << y << endl;

    swap(x,y);

    cout << "after: " << x << " " << y << endl;

    return 0;
}
```

```
double volume(int x, int y, int z)
{
    return x * y * z;
}
```

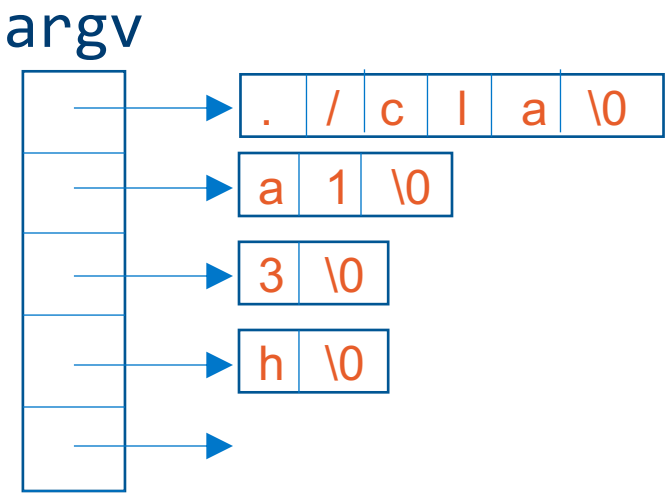
```
void swap(int &x,int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Command Line Arguments

Command Line Argument

```
1
2   #include <iostream>
3
4   using namespace std;
5
6   int main(int argc, char const *argv[])
7   {
8       for (int i = 0; i < argc; i++)
9           cout << argv[i] << endl;
10      return 0;
11  }
```

```
./cla a1 3 h hello
```



argc = 5

Command Line Argument

```
1
2   #include <iostream>
3
4   using namespace std;
5
6   int main(int argc, char const *argv[])
7   {
8       for (int i = 0; i < argc; i++)
9           cout << argv[i] << endl;
10      return 0;
11  }
```

```
./cla a1 3 h hello
```

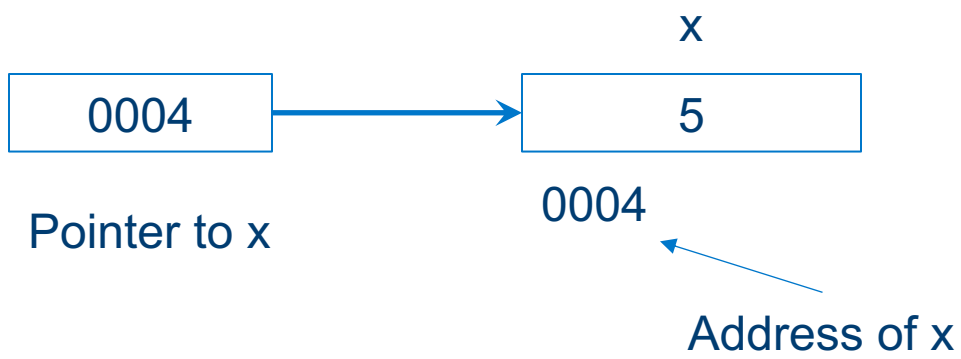
Output:

```
./cla
a1
3
h
hello
```

Pointers

Pointer Introduction

- What is a pointer?
  - A variable holding an address of a variable
- Recall:
  - Each byte in a memory has its own address
  - We can access a variable using its name or address



Pointing to and & operator

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

- Sets pointer variable p1 to "point to" int variable v1
- Operator &
  - Determines "address of" variable
- Read like:
  - "p1 equals address of v1"
  - "p1 points to v1"

Pointers and arrays

- There is close relationship between arrays and pointers in C++
- How?
  - In C++ each array name is the pointer to first element of the array
  - This pointer is constant!

That is all about this relationship

```
int  a[5];
```

- Note that it is different from

```
int *a;
```

There is no storage that a points to !!!

Pointers Arithmetic

Example: int *p, *q;

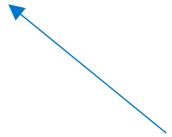
- ++/-- or adding to integer:
  - ++: add pointer to size of object that p points to (e.g. p++ adds p to sizeof(int)
- ==, >= , …
  - Two pointer should have the same type
- p-q
  - Is an integer showing the the number of objects (here, integers) and NOT byte between p and q
- Compare to nullptr/0
  - P==nullptr //by definition if it is correct it shows that p does not poin to //any variable

What is the two main usage of pointers

- Call by reference
  - C++ eliminates this need by introducing the reference concepts
  - DO NOT confuse yourselves by making relationship with reference and pointers!!

They are different !!

- Dynamic Memory Allocation



Very important !

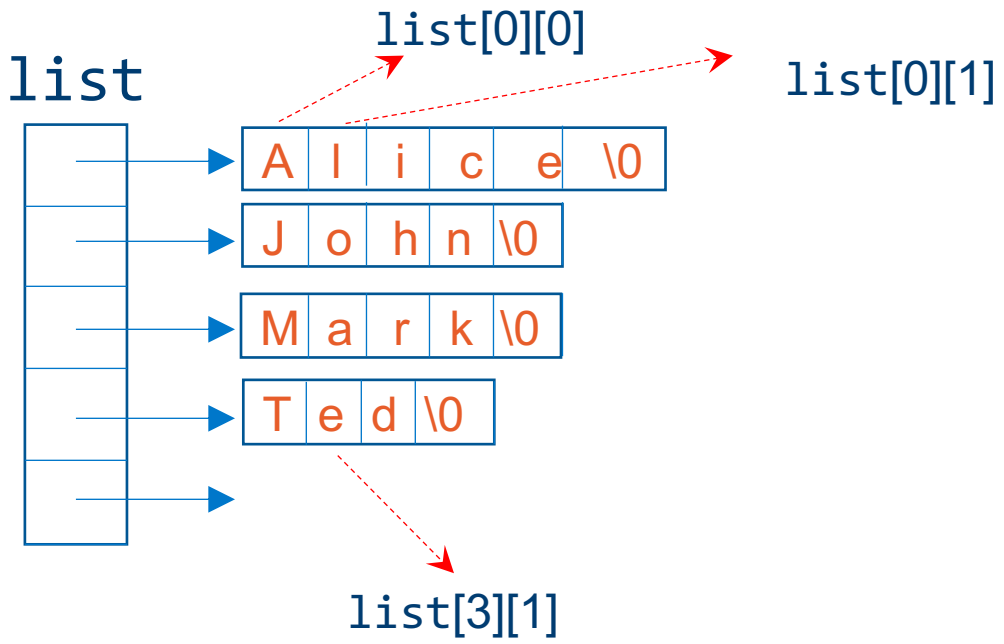
The new Operator

- Can dynamically allocate variables
  - Operator *new* creates variables and gives the its address


```
int *p;
p = new int[5];     // new allocates array of
                    // integers and returns its
                    // address (address of 1’st
                    // element)
delete [] p;     // free the memory
```

Array of Pointers

How to create such an structure? (see arr_ptr.cpp)




```
char *list[100];

// list is an array of
// pointers to char
```

Note: List[0] is like a name for the first array

Encapsulation

Object Oriented Programming

- Object Oriented Programming Foundations:
  - **Encapsulation** ✓
  - Inheritance
  - Polymorphism

Each object has

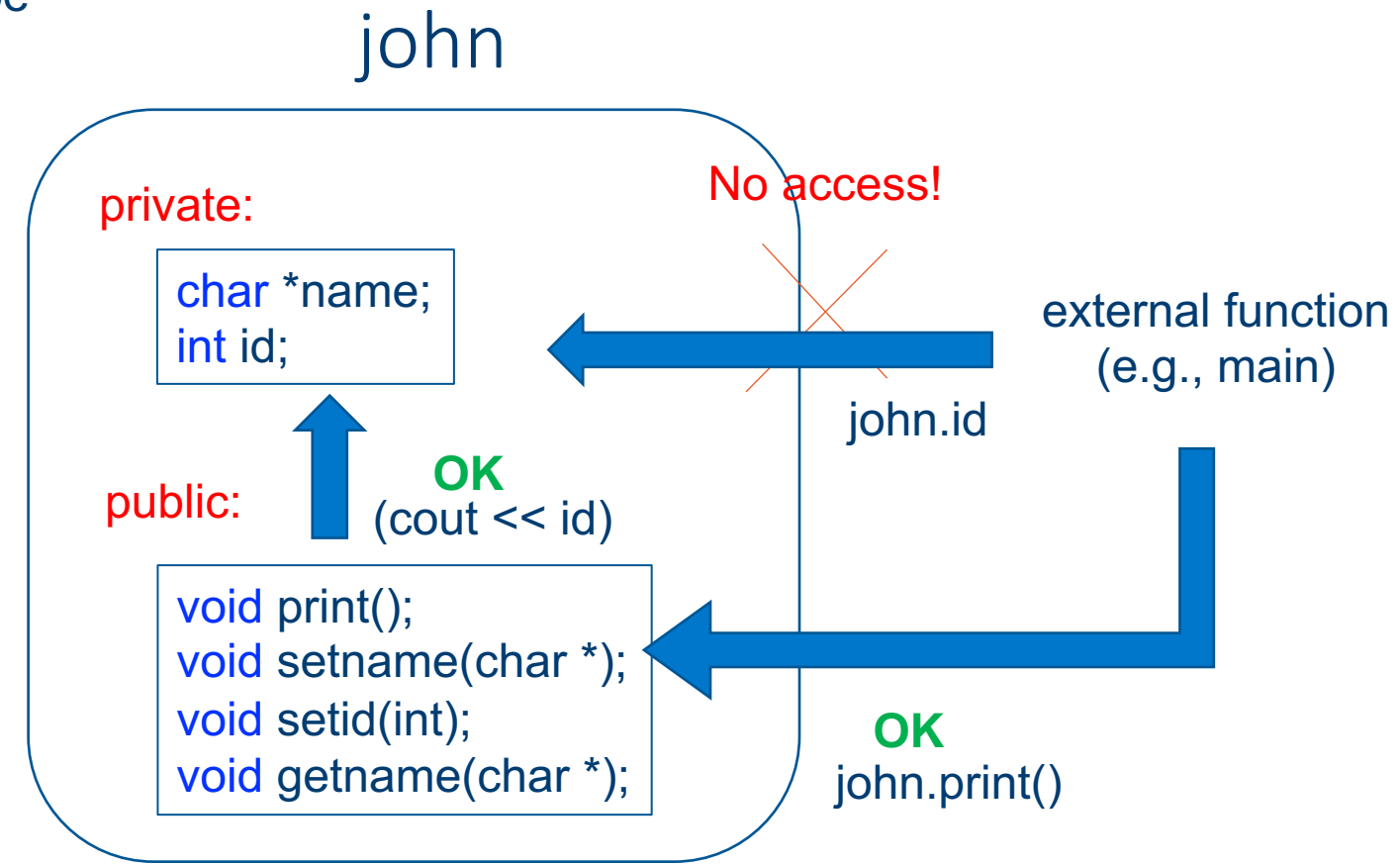
Public/Private parts

Private/Public Members

class defines a type

```
class Student{
   private:
      char *name:
      int id;
   public:
      void print();
      void setname(char *);
      void setid(int);
      void getname(char *);
};

Student john;
```

Instance of a class is an **object**;



Private/Public Members

- Private members:
  - All members in the private section can be accessed just by other member functions

- Public members:
  - All members in the public section can be accessed by any function (members or non-members)

Constructor

- Why do we need constructors?
  - Constructor provides a mechanism to take some actions automatically at the time of instantiation.

- How they are useful?
  - Object Initialization
  - Resource Allocation (e.g., Dynamic Memory Allocation)

Constructor

- When constructor is called?
  - Constructor for the object will automatically be called at the time of instantiation (no explicit call needed)

- Syntax: Constructor is a member function which :
  - Has the same name as class
  - No return value

- Why constructor for initialization?
  - Initializing an object's instance variables in a constructor ensures that the object has a well-defined state from the time of its creation.

Destructors

- Destructor provides a mechanism to take some actions at when the object lifetime is over.
  - In order to know when destructor is called, we need to know the lifetime rules.
- Syntax:
  - Name = ~ plus the name of class
  - Always no return value
  - Always no parameters

In case that several objects lifetimes are over, the order of destructor call is the reverse of creation of objects.

Scope and Lifetime Rules

- Scope:
  - Where a variable is accessible?
    - External variable: from the point that is defined till the end of file
    - Local variable: inside the block ({…}) that is defined

- Lifetime:
  - When a variable is accessible?
    - External variable: they are created at the beginning of program and exist till the end of program
    - Local variable: they are created when function is called or we enter the block ({}) and are destroyed when we return from the function or exit the block

When constructor and destructor are called?

```
39  Test a(1); //External
40
41  int main()
42  {
43      Test b(2); // 2
44
45      f(b);
46
47      return 0;
48  }
49
50  void f(Test t)
51  {
52      Test c(3);
53  }
```

```
Creating object 1
```
(1): a

```
Creating object 2
```
(2): b

```
Creating object (copy)2
```
(3): t

```
Creating object 3
```
(4): c

```
Destructing object 3
```
(5): c

```
Destructing object 2
```
(6): t

(7): b

```
Destructing object 2
```

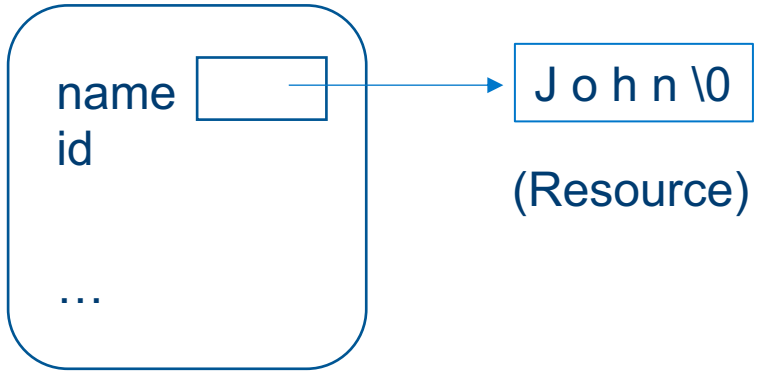
(8): a

```
Destructing object 1
```

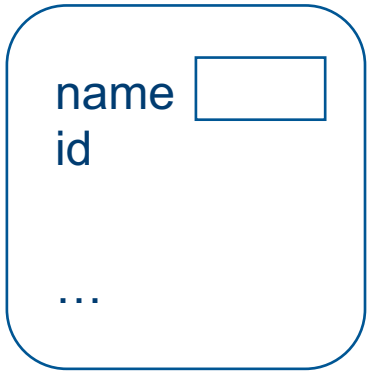
Class with Resources

- Example of resources:
  - Dynamic memory allocation
    - Allocate memory in constructor
    - Free memory in destructor

```
void Student::Student(char const *n)
{
    name = new char[strlen(n)+1];
    strcpy(name, n);
}
```

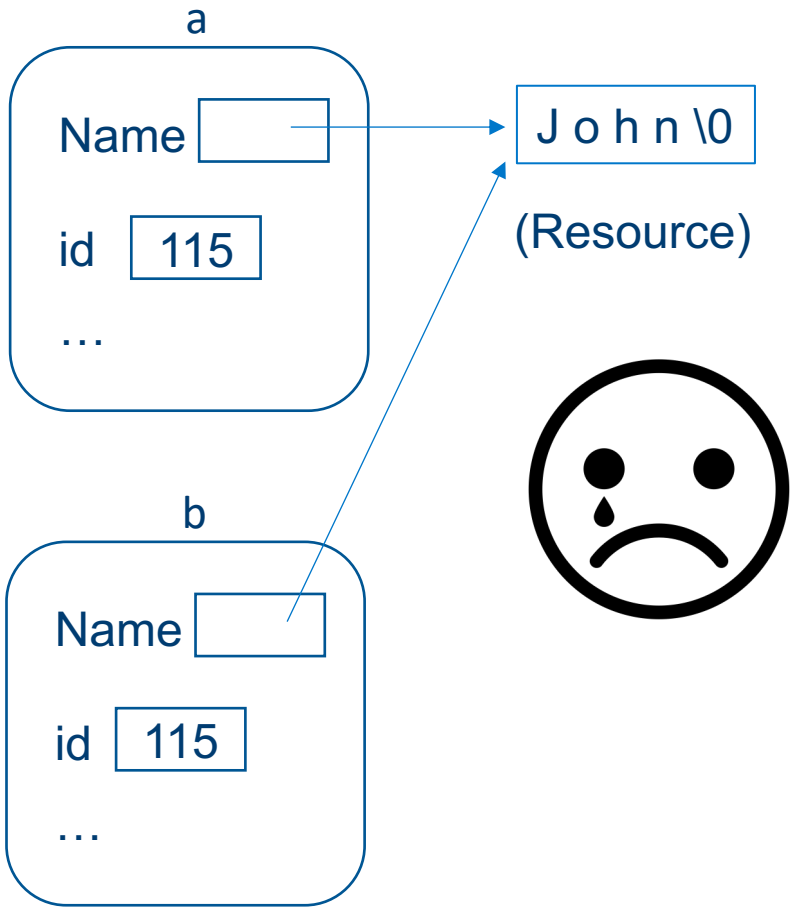



```
Student::~Student()
{
    delete [] name;
}
```



Copy Constructor for Class with Resources

- We need to do deep copy (allocating new resource and copy the information) in copy constructor.



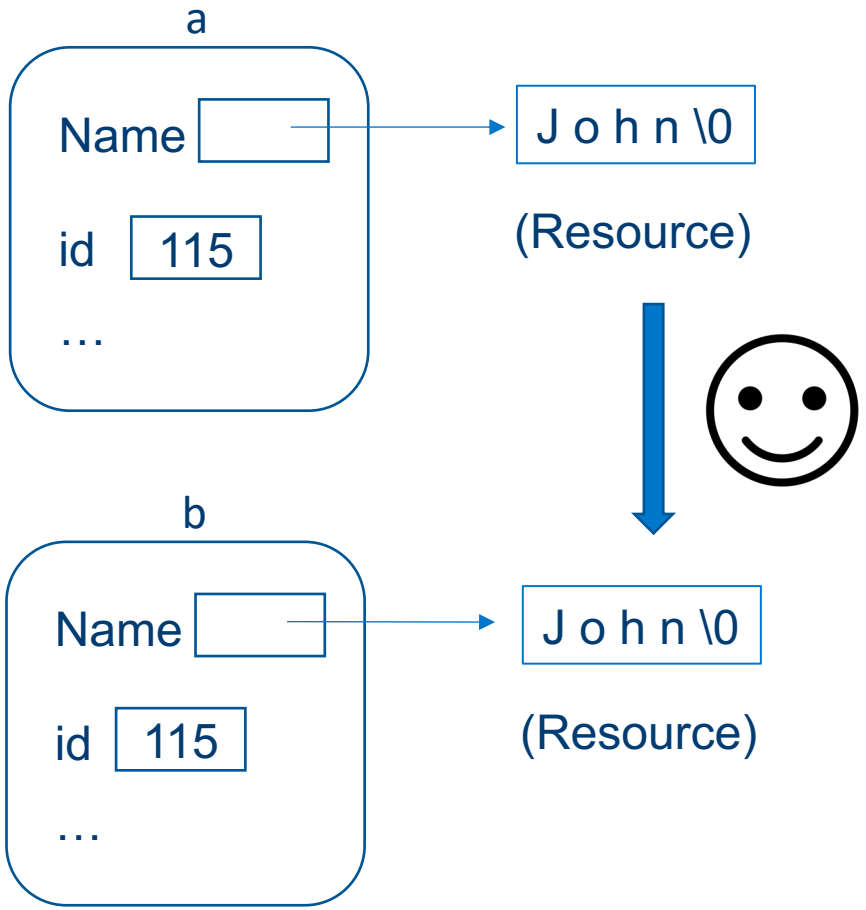

```
Student a("John", 115);
```

```
Student b=a;
```

default behavior is to copy element by element, which is not desirable !

Copy Constructor for Class with Resources

- We need to do deep copy (allocating new resource and copy the information) in copy constructor.




```
Student a("John", 115);
```

Allocate the memory and copy the information in copy constructor

```
Student b=a;
```

Copy constructor will be called for b

Static Members

- If you define a member as a static member, you just have one copy for all instances (objects) of the class

- You can class have access to the static object using the name of class rather then the name of object:

```
Student::help();

Student::num_student_obj;
```

Operator Overloading

How to Overload Operators?

```
Student x("Sarah", 115, 75.6) ;
x = y + 3;
```

- You need to define following function:
  - Helper function: `Student operator+(Student, int)`

For `y + 3` we call `operator+(y, 3)`

?

OR

  - Member function: `Student operator+(int)`

For `y + 3` we call `y.operator+(3)`

Helper or Member Function?

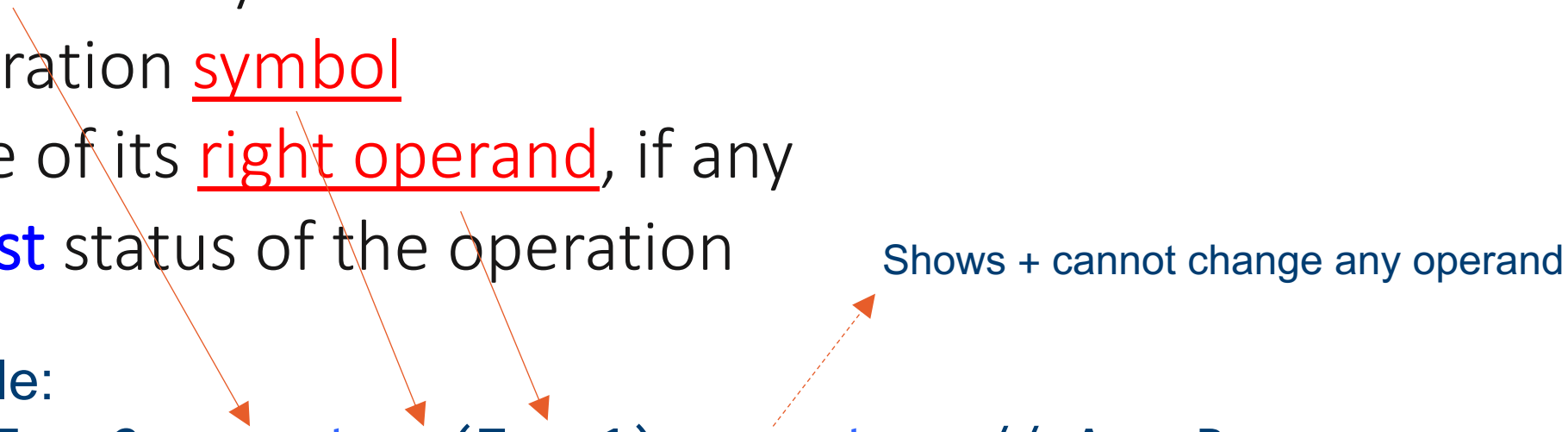
- We overload operators in either of two ways, as:
  - Member operators - part of the class definition
  - Helper operators - supporting, but outside the class definition (usually friend)

Some Limitations:

- For (assignment) operator= overloading function must be declared as a class member.
- When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.

Overloading as a Member Function

- The signature of an overloaded member operator consists of:
  - the operator keyword
  - the operation symbol
  - the type of its right operand, if any
  - the const status of the operation



Example:

```
Type2 operator+(Type1)  const ;  // A + B
Note: A+B   calls   A.operator+(B)
```

Overloading as a Member Function

Example: operator= // Assignment

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

```
Student & Student::operator=(const Student &d)
{
    id = d.id;
    grade = d.grade;
    delete [] name;

    name = new char[strlen(d.name)+1];
    strcpy(name,d.name);

    return *this;    // this the assigned object
}
```

Note:
- We do not change c
- = has side effect

```
a = c;
```

We call

```
a.operator=(c)
```

- Assign c to a (side effect)
- Returns c

Overloading as a Helper Function

- Good candidates: Those who do not change the operands
  - Example

`==` , `+` , `-`

- You have to define >> and << as helper functions and NOT member

`>>` , `<<`

(The reason is that the leftmost operand is cin
or cout and not our class type)

Overloading as a Helper Function

Example: operator<< //

friend ostream & operator<<(ostream &, const Student &);

```
ostream & operator<<(ostream & os, const Student &s)
{
    os << "\tname: " << s.name << endl;
    os << "\tID: " << s.id << endl;
    os << "\tGrade " << s.grade << endl;
    return os;
}
```

```
cout << a;
```


```
operator<<(cout, a)
```

Inheritance

Inheritance and Hierarchy

- Inheritance is the second most prominent concept next to encapsulation.

- OOP Foundations:
  - Encapsulation
  - Inheritance
  - Polymorphism

How to derive from a base class?

```
class Person{
    protected:
        char * name;

    public:
        void setname(char const *);
        void print();

        Person();                   // default constructor
        Person(char const *);
        Person(Person &);           // copy constructor

        ~Person();

};
```

Base

```
class Student : public Person{
    private:
        double grade;

    public:

        void setgrade(double);
        void printstudent();

        Student();    // default constructor
        Student(char const *, double);
        Student(Student &);  // copy constructor

        ~Student();
};
```

Derived Class

The **protected** members can be used by the members of derived class
These members cannot be accessed by non-members.

How to derive from a base class?

```
class Person{
    protected:
        char * name;

    public:
        void setname(char const *);
        void print();

        Person();                   // default constructor
        Person(char const *);
        Person(Person &);           // copy constructor

        ~Person();

};
```

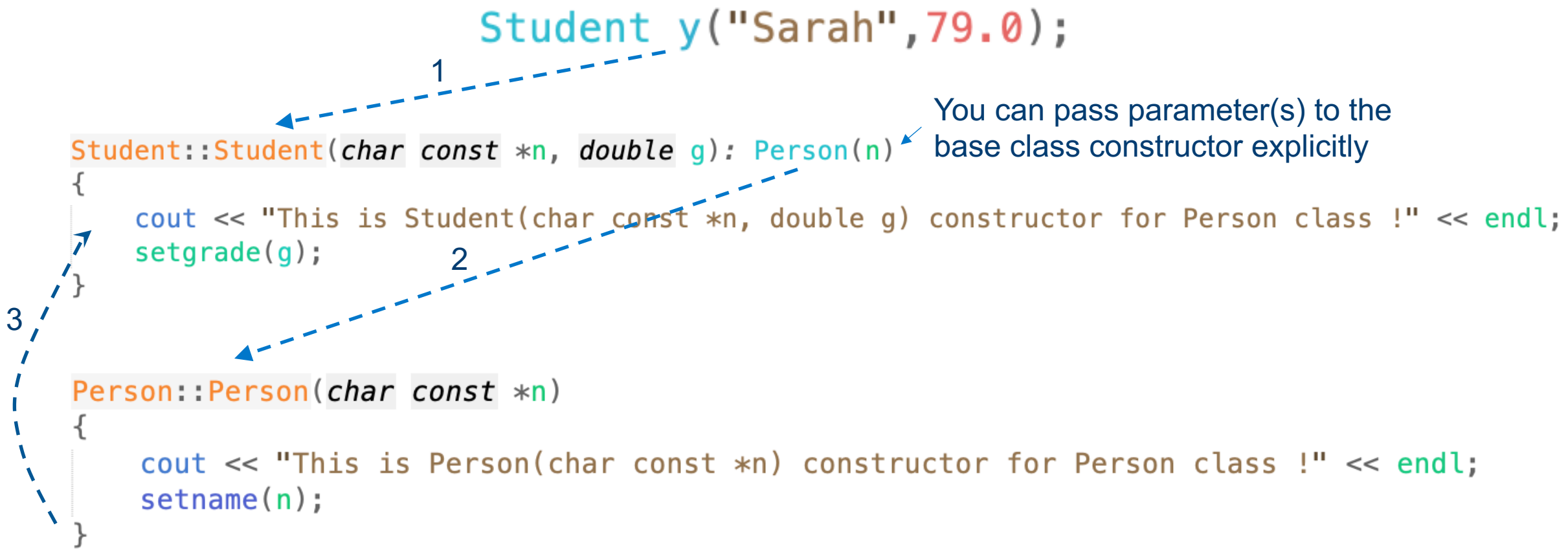
Base

```
class Student : public Person{
    private:
        double grade;

    public:

        void setgrade(double);
        void printstudent();

        Student();    // default constructor
        Student(char const *, double);
        Student(Student &);  // copy constructor

        ~Student();
};
```

Derived Class

A derived class does not by default inherit the constructors and destructors of the base class.

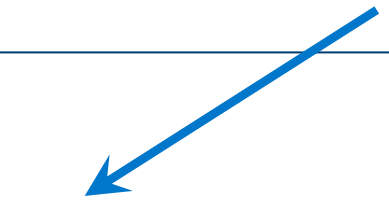
What is the order of calling constructor in base and derived class?



Initialization List in Constructors

OK

```
class Point {
   private:
      const int x;
      const int y;
   public:
      Point(int  = 0, int  = 0);

      int getX() const {return x;}
      int getY() const {return y;}
};
Point::Point(int i , int j ):x(i), y(j)
{
 }
```

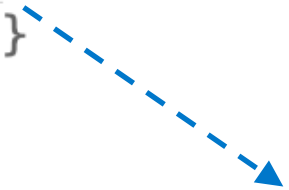


NOT OK

```
class Point {
   private:
      const int x;
      const int y;
   public:
      Point(int  = 0, int  = 0);

      int getX() const {return x;}
      int getY() const {return y;}
};
Point::Point(int i , int j ):x(i), y(j)
{
   x = i;
   y = j;
}
```

What is the order of calling destructor in base and derived class?

```
Student::~Student()
{
    // The based constructor will be called after the based constructor
    cout << "This is the Student destructor !" << endl;
}
```



Derive class destructor automatically calls the based class destructor after finishing the job.

```
Person::~Person()
{
    cout << "This is the Person destructor !" << endl;
    if(name)
        delete [] name;
}
```

What is the order of calling destructor in base and derived class?

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.printstudent();

    return 0;
}
```

```
This is Person(char const *n) constructor for Person class !
```

```
name: John
```

```
This is Person(char const *n) constructor for Person class !
This is Student(char const *n, double g) constructor for Person class !
```

```
name: Sarah
grade: 79
```

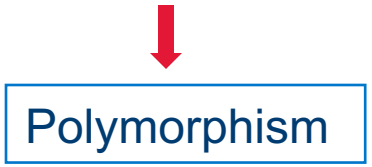
```
This is the Student destructor !
This is the Person destructor !
```

```
This is the Person destructor !
```

Polymorphism

Central Question

- What will happen if we define a function with the same name (identifier) that already exists in the base class?
  - Shadowing (default)
  - Overriding (when this function is virtual in base class) => next week



Shadowing

- A member function of a derived class shadows the base class member function with the same identifier.

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Student("Jessi",87.8);
    p-> print();

    return 0;
}
```

```
void Person::print()
{
    if(name)
        cout << "\tname: " << name << endl;
    else
        cout << "\tThis is an empty object !" << endl;

}
```

Shadow

```
void Student::print()
{
    Person::print();
    if(name)
        cout << "\tgrade: " << grade << endl;
}
```

Shadowing

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Student("Jessi",87.8);
    p-> print();

    delete p;

    return 0;
}
```

```
This is Person(char const *n) constructor for Person class !
This is Student(char const *n, double g) constructor for Person class !
name: Jessi
This is the Person destructor !
This is the Student destructor !
This is the Person destructor !
This is the Person destructor !
```

Overriding

```
class Person{
    protected:
        char * name;

    public:
        void setname(char const *);
        virtual void print();

        Person();                   // default constructor
        Person(char const *);
        Person(Person &);           // copy constructor

        virtual ~Person();
};
```

Solution: **virtual** functions can be override by the derived class

Overriding

```
int main()
{
    Person x("John");
    x.print();

    Student y("Sarah",79.0);
    y.print();

    Person * p = new Student("Jessi",87.8);
    p-> print();

    delete p;

    return 0;
}
```

```
This is Person(char const *n) constructor for Person class !
This is Student(char const *n, double g) constructor for Person class !
name: Jessi
grade: 87.8
This is the Student destructor !
This is the Person destructor !
This is the Student destructor !
This is the Person destructor !
This is the Person destructor !
```

Abstract Class

```
class Account{
    private:
        double balance; // data member that stores the balance

    protected:
        double getBalance() const; // return the account balance
        void setBalance( double ); // sets the account balance

    public:
        Account( double = 0.0); // constructor initializes balance

        virtual void credit(double);

        virtual bool debit(double);

        virtual void display(ostream &) const = 0;
};
```

Pure virtual Function

Special Topics

- String in C (c-style string): in C, we don’t have a built-in string type !
  - We use array of char and \0 (null) to store our string

```
char a[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
char b[] = “Hello”;
```

- Useful function are available in #include<cstring>
  - Examples: strcpy, strcat

Special Topics

- In C++, this problem is solved by defining a **`string`** class

- Useful function are available in #include<string>

Read:

https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1132/handouts/08-C++-Strings.pdf

Write the Student class with **`name`** as a **`string`**

Template Function/Class

Motivation

Overloading

```
void swapValues(int& var1, int& var2)
{
        int temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

```
void swapValues(char& var1, char& var2)
{
        char temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

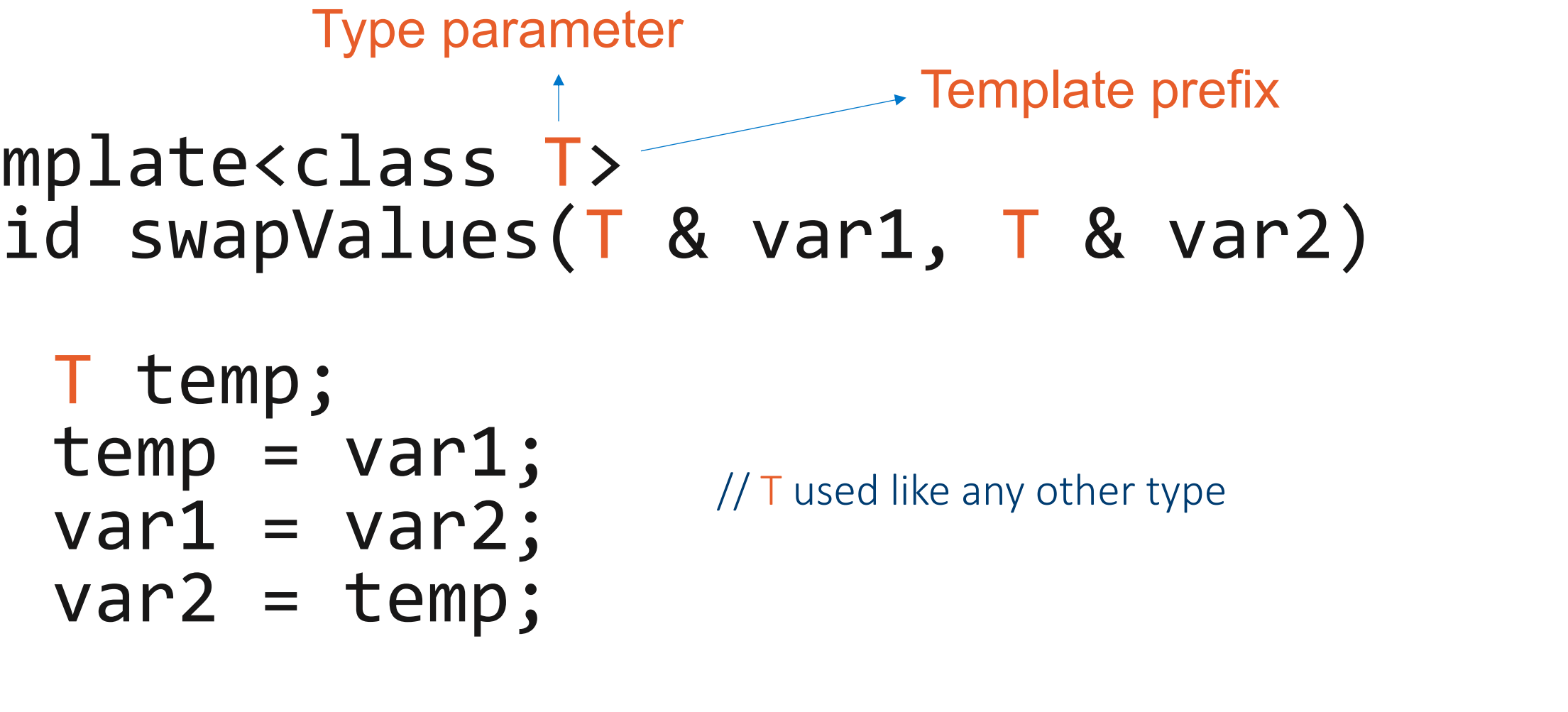
Works for variable type **int**

Works for variable type **char**

Can we do the better job?

Note: the codes of two functions have identical logic!

Function Template



Function Template

```
int main()
 {
      int x = 1, y = 2;

      cout << "x = " << x << ", y = " << y <<endl;

      swapValues(x, y); // swapValues(int, int) will be called

      cout << "x = " << x << ", y = " << y <<endl;

      return 0;
 }
```

How to create a class template?

```
// This is written for the integer pairs  !
class Pair
{
      public:
            Pair();
            Pair(int firstVal, int secondVal);
            void setFirst(int newVal);
            void setSecond(int newVal);
            int getFirst() const;
            int getSecond() const;

      private:
            int first; int second;
};
```

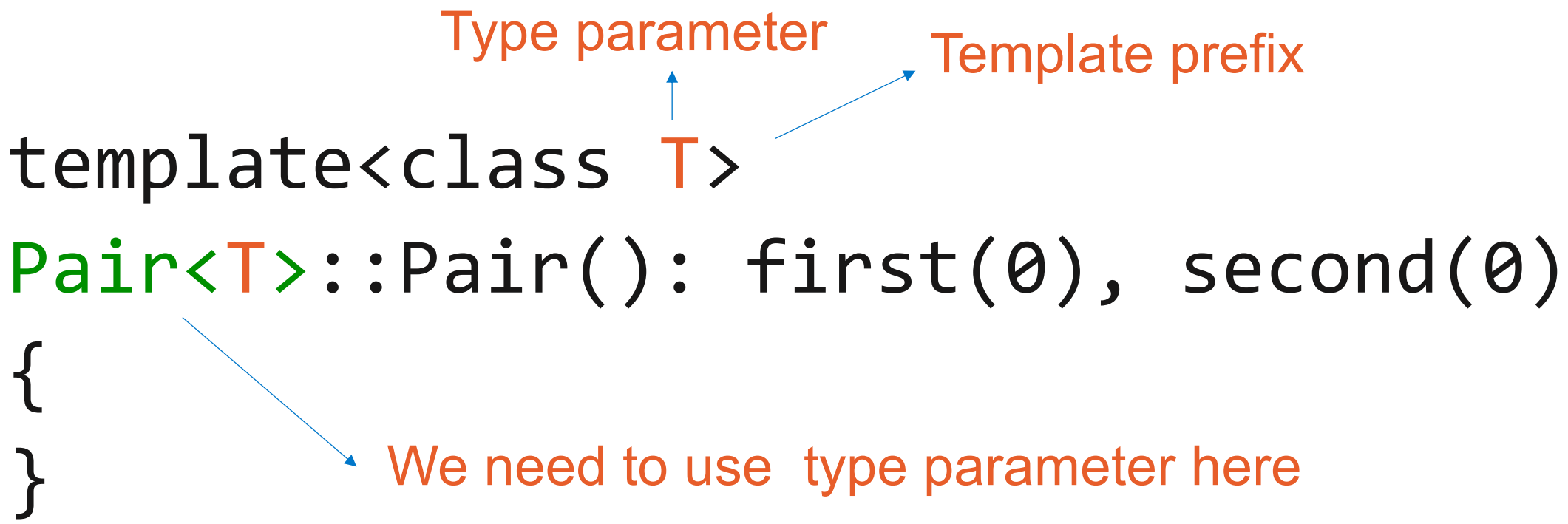
Class Template

Template prefix

```
template<class T>
class Pair
{
      public:
            Pair();
            Pair(T firstVal, T secondVal);
            void setFirst(T newVal);
            void setSecond(T newVal);
            T getFirst() const;
            T getSecond() const;
      private:
            T first; T second;
};
```

From: Absolute C++

How to define a member function of a class template?



(Default Constructors)

How to create the objects of a class template?

```
int main()
{
  Pair<int> x;
  Pair<int> y(2,3);

  cout << x.accessFirst()<< endl;
  cout << x.accessSecond()<< endl;
  cout << y.accessFirst()<< endl;
  cout << y.accessSecond()<< endl;

  return 0;

}
```

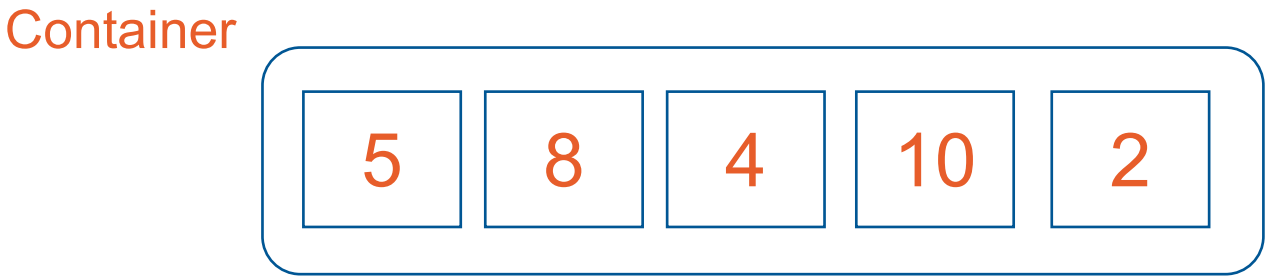
We need the type parameter when creating the objects

Containers in STL

What is a Container?

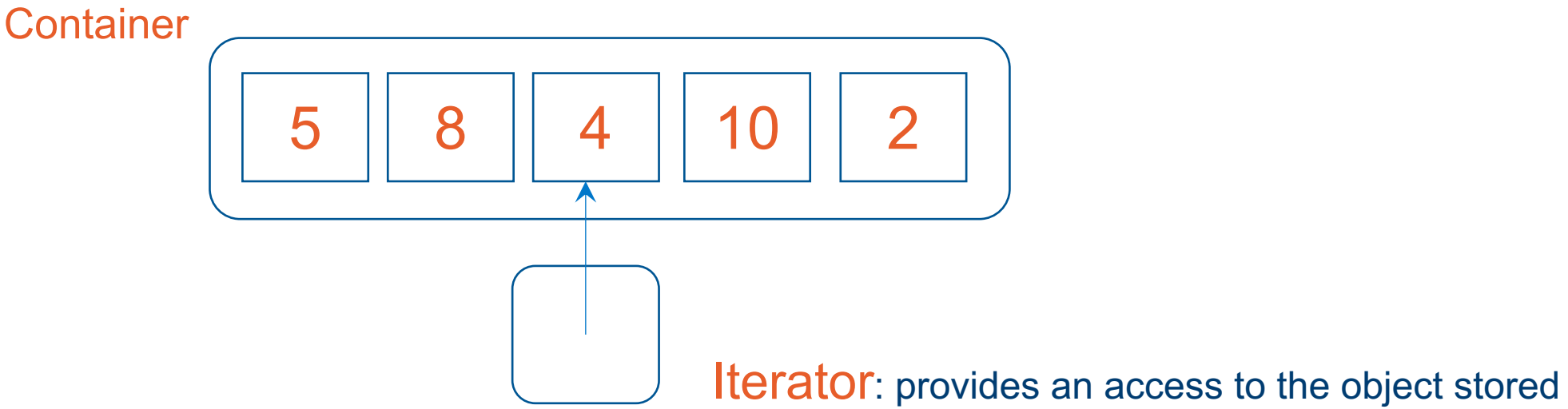
- A **container** is an object that holds other objects.
  - Example: vector class defined in STL

The Standard Template Library



What is iterator?

- An **iterator** is an object used to specify a “position” in a container.
  - All iterators can be incremented to move the position towards the end by one or dereferenced to fetch the value in the container to which the iterator refers.



How to Manipulate Iterators?

++ :advance the iterator to the next data item in a container

-- : moving the iterator to the previous data item in a container

== : test if two iterators point to the same data item in a container

!=

* : if p is an iterator variable *p give the access to data point p showing the location

Using Iterators and Containers

- C++ STL containers (both sequential and associative) define "helper classes," called iterators, to help iterate over each item in the container.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v;

    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    std::vector<int>::iterator i;
    for (i = v.begin(); i != v.end(); ++i)
    {
        std::cout << *i << std::endl;
    }
    return 0;
}
```

Container (vector)

Iterator definition

Dereference operator is used to have access to the objects in the container

vector
Container Example

Special Members of iterators

If c is a container, then

- c.begin()
  - returns an iterator that refers to the first data item in the container.
- c.end()
  - returns an iterator that refers to a position beyond the end of the container.

Kinds of Iterators

- Random Access:

  ++ , -- , == , != , * , random access

- Bidirectional

  ++, --, ==, != , *

- Forward

  ++, ==, != , *

Random Access Iterators	stronger than	Bidirectional Iterators	stronger than	Forward Iterators

Note that different containers have different kinds of iterators (see slide 20):

Which iterators I can use with a container?

- Each container class has "own" iterator type
  - Similar to how each data type has own pointer type

std::vector<int>::iterator

Random Access

std::list<int>::iterator

Bidirectional Access

Constant Iterator

```cpp
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<string> names;
    names.push_back("John");
    names.push_back("Amanda");

    list<string>::const_iterator i;
    for (i = names.cbegin(); i != names.cend(); ++i)
    {
        cout << *i << endl;
    }
    return 0;
}
```

Container (`list`)

const iterator

Reverse Iterator

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<string> names;
    names.push_back("john");
    names.push_back("amanda");

    list<string>::reverse_iterator i;
    for (i = names.rbegin(); i != names.rend(); ++i)
    {
        cout << *i << endl;
    }
    return 0;
}
```

Container (`list`)

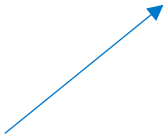
reverse iterator

Types of Containers

- Sequential Containers:
  - vector
  - list
  - deque
- The Container Adapters
  - stack
  - queue
- Associative Containers
  - set
  - map

Map Example

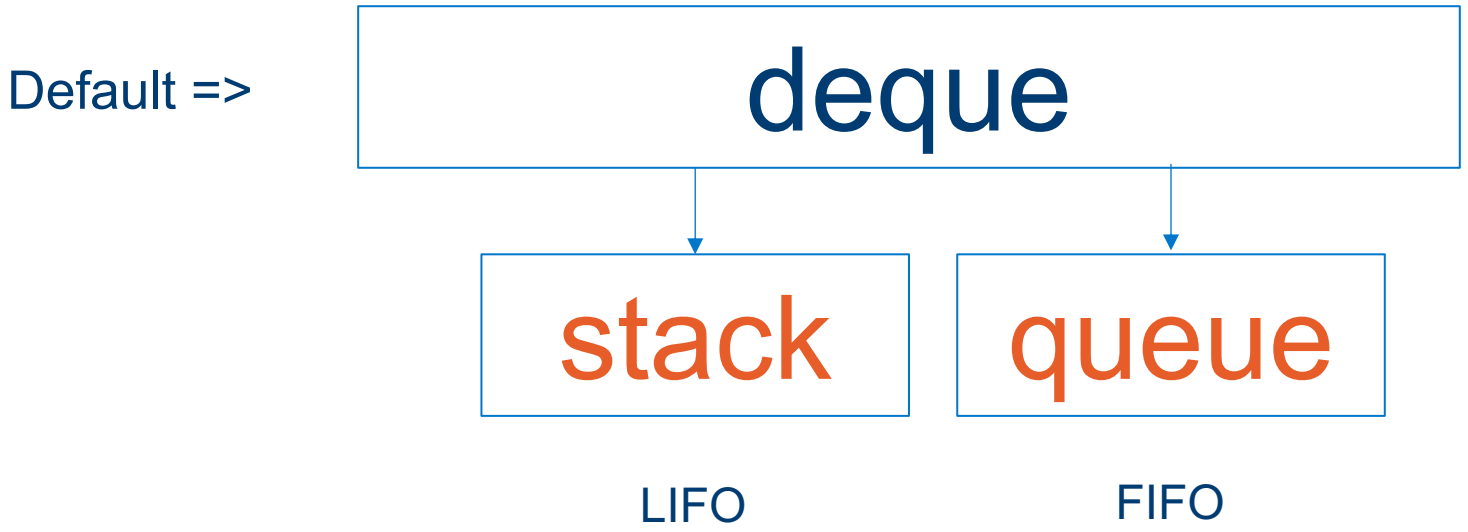
```
#include <map>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    map<string, int> super_heros;
    super_heros["batman"] = 32;
    super_heros["wolverine"] = 137;
    super_heros["jean gray"] = 25;
    super_heros["superman"] = 35;

    map<string, int>::iterator i;
    for (i = super_heros.begin(); i != super_heros.end(); ++i)
    {
        cout << "Age of " << i->first << " is " << i->second << endl;
    }

    return 0;
}
```



Can use [ ] notation to access the map

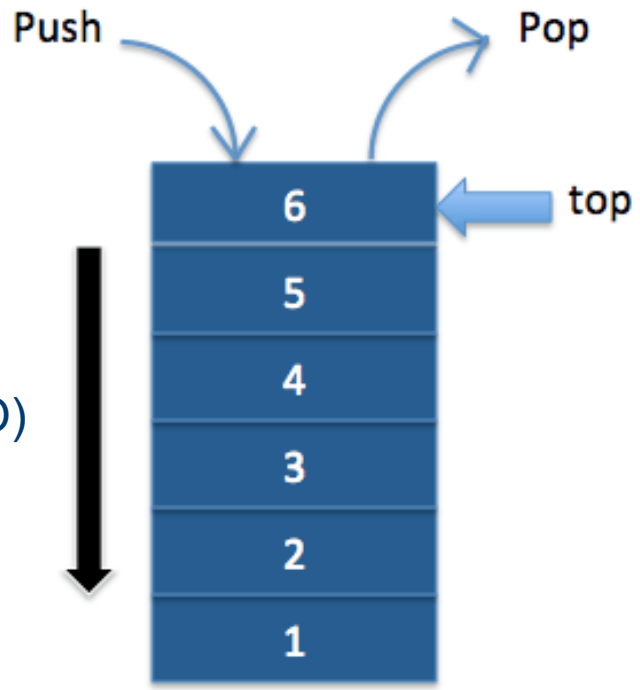
Container Adapters stack and queue

- Example: stack template class by default implemented on top of deque template class:

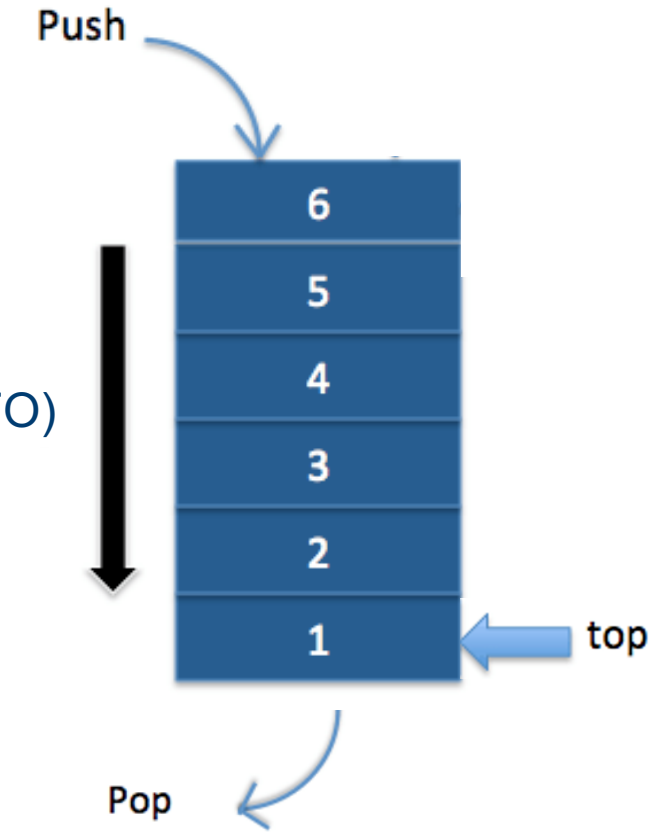


Stack and Queue

Last In First Out (LIFO)



First In First Out (FIFO)



Generic Function in STL

Generic Algorithms in STL

- The STL supplies a large set of generic algorithms that operate on containers:
- Hence, a generic algorithm may operate on any data structure that provides an iterator type that meets the iterator requirements of that algorithm.
- Parameters in a function call are iterators, not containers

```
#include <algorithm>
```

Case Study (count)

```
template <class ForwardIterator, class T>
int count(ForwardIterator first, ForwardIterator last, const T & target);
```

Description:

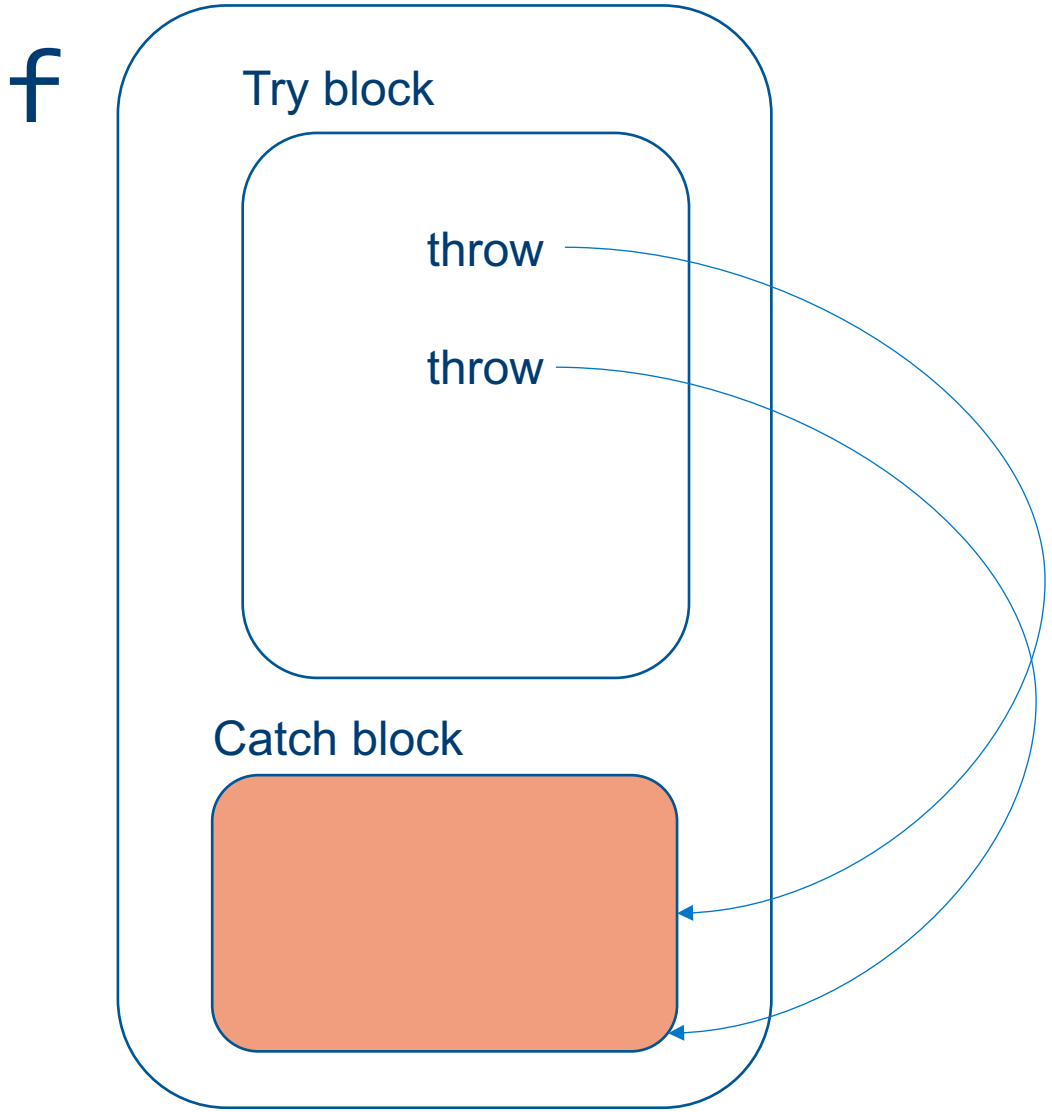
- Traverse the range **[first, last)** and returns the number of elements equal to target.

```
vector<char> v = {'X', 'A', 'N', 'B', 'X', 'Z'};
cout << count(v.begin(),v.end(), 'X') << endl;
```

```
Count of X is : 2
```

Exceptions

Why Exception?



Error handling codes are concentrated in the catch block

You can catch the exception in side the function which called or in caller function

Exception Class Example

```
class NoMilk
{
public:
    NoMilk( ) {}
    NoMilk(int howMany) : count(howMany) {}
    int getCount( ) const { return count; }
private:
    int count;
};
```

Exception Class for Toy Example

```
int main( )
{
    int donuts, milk;
    double dpg;
    try
    {
        cout << "Enter number of donuts: ";
        cin >> donuts;
        cout << "Enter number of glasses of milk: ";
        cin >> milk;

        if (milk <= 0)
            throw NoMilk(donuts);

        dpg = donuts / double(milk);
        cout << donuts << " donuts.\n";
        cout << milk << " glasses of milk.\n";
        cout << "You have " << dpg;
        cout << " donuts for each glass of milk.\n";
    }
    catch(NoMilk e)
    {
        cout << e.getCount( ) << " donuts, and No Milk!\n" << "Go buy some milk.\n";
    }
    cout << "End of program.\n";

    return 0;
}
```

Invokes constructor of NoMilk class

Throwing Exception in Function Example

function throws DividebyZero exception

```
try
{
      quotient = safeDivide(num, den);
}
catch (DivideByZero)
{
  …
}
```

We can handle error in a caller.

See: exception-function1.cpp and exception-function2.cpp and