Comp151

C++:

Data Abstraction & Classes

Topics in Today's Lecture

- Data Abstraction
- Classes: Name Equivalence vs. Structural Equivalence
- Classes: Restrictions on Data Members
- Classes: Location of Function declarations/definitions
- Classes: Member Access Control
- Implementation of Class Objects

A Brief History of C++

- Bjarne Stroustrup of AT&T Bell Labs extends C with Simula-like classes in the early 1980s; the new language was called "C with Classes".
- C++, the successor of "C with Classes", was designed by Stroustrup in 1986; version 2.0 was introduced in 1989.

The design of C++ was guided by the following three principles:

1. The use of classes should not result in programs executing any more slowly than programs not using classes.

2. C programs should run as a subset of C++ programs.

3. No run-time inefficiency should be added to the language.

Data Abstraction

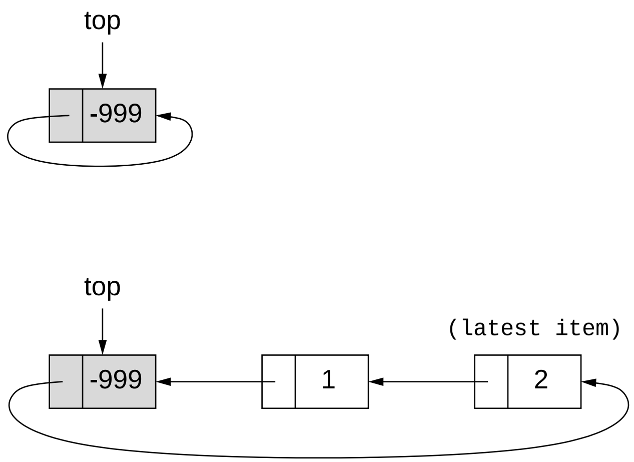
- What is a "car"?
- What is a "stack"?

A data abstraction is a simplified view of an object that includes only features one is interested in while hiding away the unnecessary details.

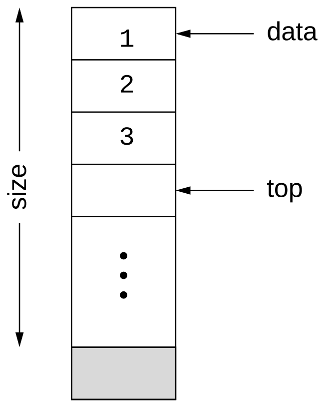
In programming languages, a data abstraction becomes an abstract data type or a user-defined type.

In OOP, it is implemented as a class.

Example: Implement a Stack with a Linked List [comp151]



Example: Implement a Stack with an Array [comp151]



Example: stack_ar.h [comp151]

data members

access control

```
class Stack
{
  private:
    int  size;          // max. size of storage
    int* top;           // point to the next available space
    int* data;          // data storage

  public:
    Stack( int N);      // a constructor
    ~Stack();           // destructor

    // basic operations
    void push( int x ); // add another datum
    int  pop( );        // get the most recent datum

    // status operations
    int  num_elements( ) const;
    bool empty( ) const;
    bool full( ) const;
};
```

member functions (public interface)

Information Hiding [comp151]

- An *abstract specification* tells us the behavior of an object independent of its implementation. i.e. It tells us *what* an object does independent of *how* it works.
- Information hiding is also known as *data encapsulation*, or *representation independence*.
- *The principle of information hiding*:
  Design a program so that the implementation of an object can be changed without affecting the rest of the program.
  e.g. Changing the implementation of a stack from an array to a linked list has no effect on users' programs.

Class Name: Name Equivalence

A class definition introduces a new abstract data type. C++ relies on *name equivalence* NOT *structure equivalence* for class types. The program below works:

```
#include <iostream.h>
class X {public: int a;};
void main()
{
  X x1, x2;
  x1.a = 1;
  x2.a = 2;

  cout << " x1.a = " << x1.a << ", x2.a = " << x2.a << endl;
  x2 = x1;
  cout << " x1.a = " << x1.a << ", x2.a = " << x2.a << endl;
}
```

Example: stack_ll.h

data members

access control

```
class  Stack
{
private:
   struct   node
    {
            int     item;
            node    *next;
    };
   node     *top;                  // points to top of stack
   int      size;                  // max. size of storage

public:
   Stack(  int  N);                // a constructor
   ~Stack();                       // destructor

   // basic operations
   void   push( int  x );          // add another datum
    int   pop( );                  // get the most recent datum

   // status operations
    int  num_elements( )   const;
   bool   empty( )   const;
   bool   full( )   const;
};
```

member functions (public interface)

Class Data Members

Data members can be any basic type, or any user-defined types that have already been "seen" (defined).

- A class name can be used (but only as a pointer) in its own definition:

  class node { public: int info; node *next;};

- It can also be used as a forward declaration for class pointers:

```
class Node;                       // forward declaration
class Stack
{
     int size;
     Node* top;           // points to an object with forward declaration
     Node x;                          // Error: Node  not defined!
};
```

Class Name: Name Equivalence(cont)

A class definition introduces a new abstract data type. C++ relies on *name equivalence* NOT *structure equivalence* for class types. The program below does not compile:

```
#include <iostream.h>
class X {public: int a;};
class Y {public: int a;};
void main()
{
  X x; Y y;
  x.a = 1;
  y.a = 2;

  cout << " x.a = " << x.a << ", y.a = " << y.a << endl;
  y = x;
  cout << " x.a = " << x.a << ", y.a = " << y.a << endl;
}
```

Class Member Functions

These are the functions *declared* inside the body of a class (they can be either public or private). They can be *defined* in two ways:

- within the class body, in which case, they are *inline* functions.

```
class Stack
{   ...
    void push(int x) { *top = x; ++top; }
    int pop() { --top; return (*top); }
};
```

Class Data Members ..

But, they can NOT be initialized inside the class definition.
The program below will not compile.

```
class X {public: int a=1;};
void main()
{
  X x;
  cout << " x.a = " << x.a << endl;
}
```

Initialization should be done with appropriate constructors, or member functions.

Member Access Control

A member of a class can be:

public : accessible to anybody

private : accessible only to member functions and friends of the class. ⇒ Enforce information hiding

protected : accessible to member functions and friends of the class, as well as to member functions and friends of its derived classes (subclasses). We will discuss this in greater detail later in the course.

Class Member Functions ..

- outside the class body

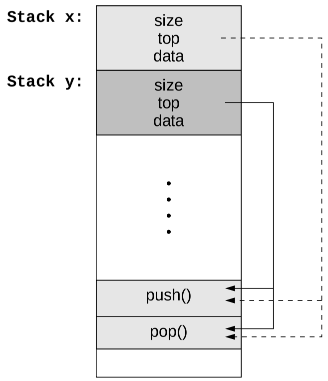
```
class Stack
{   ...
    void push(int x);
    int pop();
};

void Stack::push(int x) { *top = x; ++top; }
int Stack::pop() { --top; return (*top); }
```

Question: Can we add data and function declarations to a class after the end of the class definition?

How Are Objects Implemented?

- Each class object gets its own copy of the class data members.
- All objects of the same class share *one* copy of the member functions.




```
int main()
{
    Stack x(2), y(3);

    x.push(1);
    y.push(2);
    y.pop();
}
```

Example: Member Access Control


```
class Stack
{
  private:
    Cell* top;
  public:
    int size;
  ...
};

int main()
{
    Stack x;
    cout << x.size;          // OK: size is public
    x.push(2);               // OK: push() is public
    cout << x.top->info;     // ERROR: cannot access top
}
```