Object-Oriented Programming and Data Structures

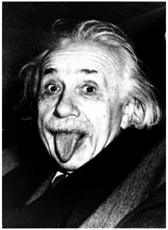
COMP2012: Data Abstraction & Classes

Prof. Brian Mak
Prof. C. K. Tang

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Part I

What is Data Abstraction?



$\Longrightarrow$

Data Abstraction: What is a Chair?





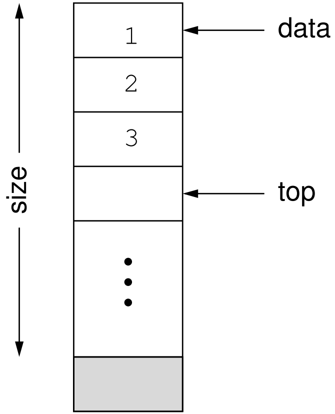



Data Abstraction: What is a Stack?

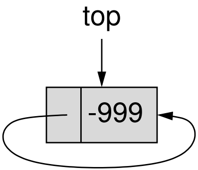


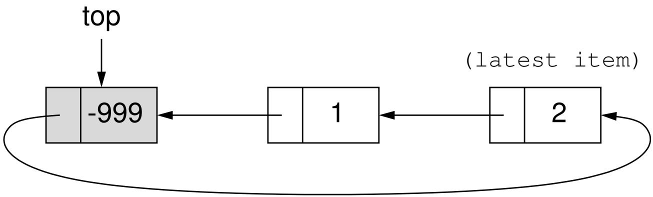

- A data abstraction is a simplified view of an object that includes only features one is interested in while hides away the unnecessary details.
- In programming languages, a data abstraction becomes an abstract data type or a user-defined type.
- In OOP, it is implemented as a class.

Example: Implement a Stack with an Array



Example: Implement a Stack with a Linked List





Information Hiding

- An abstract specification tells us the behavior of an object independent of its implementation.
- 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.

```
#include <iostream>                                          /* File: stack.h */
#include <cstdlib>
using namespace std;
const int BUFFER_SIZE = 5;

class Stack
{
  private:
    int data[BUFFER_SIZE];                           // Use an array to store data
    int top_index;                               // Starts from 0; -1 when empty

  public:
    // CONSTRUCTOR member functions
    Stack( );                                             // Default constructor
    // ACCESSOR member functions: const => won't modify data members
    bool empty( ) const;                             // Check if the stack is empty
    bool full( ) const;                               // Check if the stack is full
    int size( ) const;                  // Give the number of data currently stored
    int top( ) const;                             // Retrieve the value of the top item
    // MUTATOR member functions
    void push(int);                            // Add a new item to the top of the stack
    void pop( );                                  // Remove the top item from the stack
};
```

Structure vs. Class

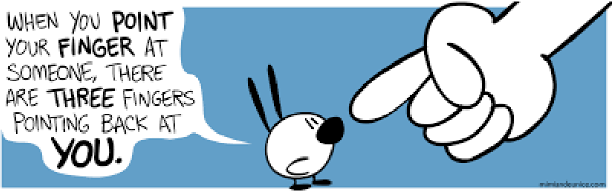
```
const int BUFFER_SIZE = 5;                              /* File: stack-struct.h */

struct Stack
{
    int data[BUFFER_SIZE];                       // Use an array to store data
    int top_index;                               // Starts from 0; -1 when empty

    Stack( );                                    // Default constructor
    bool empty( ) const;                         // Check if the stack is empty
    bool full( ) const;                          // Check if the stack is full
    int size( ) const;                           // Give the number of data currently stored
    int top( ) const;                            // Retrieve the value of the top item
    void push(int);                              // Add a new item to the top of the stack
    void pop( );                                 // Remove the top item from the stack
};
```

- In C++, structures are special classes and they can have member functions.
- By default,

struct { ... }; ≡ class { public: ... };
class { ... }; ≡ struct { private: ... };

Part II

C++ Class Basics & this Pointer



Class Name: Name Equivalence

- A class definition introduces a new abstract data type.
- C++ relies on name equivalence (and not structure equivalence) for class types.

```
class X { int a; };
class Y { int a; };
class W { int a; };
class W { int a; };                    // Error, double definition

X x;
Y y;

x = y;                                 // Error: type mismatch
```

Class Data Members

Data members can be any basic type, or any user-defined types if they are already declared.

Below are special cases:

- A class name can be used in its own definition for its pointers:

  ```
  class Cell { int info; Cell *next; ... };
  ```

- A forward declaration for class pointers:

  ```
  class Cell;                                  // Forward declaration
  class Stack
  {
      int size;
      Cell* data;   // Points to an object with forward declaration
      Cell x;                        // Error: Cell not defined yet!
  };
  ```

Data Members Cannot be Initialized In Class Definition

```
class Stack
{
    ...
    // Error: data member cannot be initialized
    // inside class definition
    int top_index = 0;
};
```

Initialization should be done with appropriate

- constructors, or
- member functions

of the class.

Class Member Functions

- These are the functions declared inside the body of a class.
- They can be defined in two ways:

1. Within the class body, then they are inline functions. The keyword inline is optional in this case.

```
class Stack
{
    ...
    void push(int x) { if (!full( )) data[++top_index] = x; }
    void pop( ) { if (!empty( )) --top_index; }
};
```

Or,

```
class Stack
{
    ...
    inline void push(int x) { if (!full( )) data[++top_index] = x; }
    inline void pop( ) { if (!empty( )) --top_index; }
};
```

2. Outside the class body, then add the prefix consisting of the class name and the class scope operator ::
   (Any benefits of doing this?)

```
/* File: stack.h */
class Stack
{
    ...
    void push(int x);
    void pop( );
};
```

```
/* File: stack.cpp */
void Stack::push(int x) { if (!full( )) data[++top_index] = x; }
void Stack::pop( ) { if (!empty( )) --top_index; }
```

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

Class Scope and Scope Operator ::

- C++ uses lexical (static) scope rules: the binding of name occurrences to declarations are done statically at compile-time.
- Identifiers declared inside a class definition are under its scope.
- To define the members functions outside the class definition, prefix the identifier with the class scope operator ::
- e.g., Stack::push, Stack::pop

```
int height;
class Weird
{
    short height;
    Weird( ) { height = 0; }
};
```

Q1: Which "height" is used in Weird::Weird()?

Q2: Can we access the global height inside the Weird class body?

Inline Functions

- Function calls are expensive because when a function is called, the operating system has to do a lot of things behind the scene to make that happens.

```
int f(int x) { return 4*x*x + 9*x + 1; }
int main( ) { int y = f(5); }
```

- For small functions that are called frequently, it is actually more efficient to unfold the function codes at the expense of program size (both source file and executable).

```
int main( ) { int y = 4*5*5 + 9*5 + 1; }
```

Inline Functions ..

- But functions has the benefit of easy reading, easy maintenance, and type checking by the compiler.
- You have the benefits of both by declaring the function inline.

```
inline int f(int x) { return 4*x*x + 9*x + 1; }
int main( ) { int y = f(5); }
```

- When you define a member function inside a class, it is treated as an inline function.
- *However*, C++ compilers may not honor your inline declaration.
- The inline declaration is just a hint to the compiler which still has the freedom to choose whether to inline your function or not, especially when it is large!

Inline Class Member Functions

- Class member functions can be defined inside the class body and are automatically treated as inline functions.
- To enhance readability, one may also define them outside the class definition but in the same header file.

``/* File: stack1.h */``` ```class Stack``` ```{``` ``` ...``` ``` inline void pop( )``` ``` {``` ``` if (!empty( ))``` ``` --top_index;``` ``` }``` ```};```	```/* File: stack2.h */``` ```class Stack``` ```{``` ``` ...``` ``` inline void pop( );``` ```};``` ```inline void Stack::pop( )``` ```{``` ``` if (!empty( )) --top_index;``` ```}```

Member Access Control

A member of a class can be:

1. public: accessible to anybody (class developer and application programmers)
2. private: accessible only to
   - member functions and
   - friends of the class

   $\Rightarrow$ class developer enforces information hiding
3. protected: accessible to
   - member functions and friends of the class, as well as
   - member functions and friends of its derived classes (subclasses)

   $\Rightarrow$ class developer restricts what subclasses may directly use (more about this when we talk about inheritance)

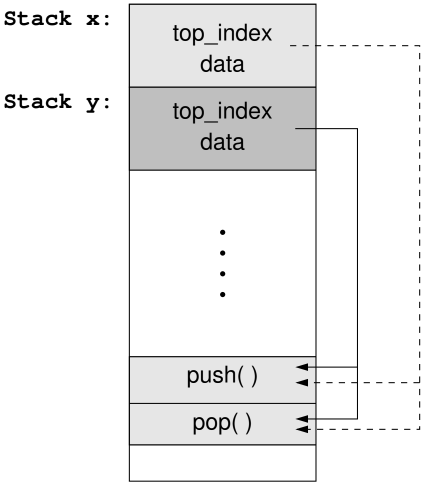
Example: Member Access Control

```
class Stack
{
  private:
    int data[BUFFER_SIZE];
    int top_index;
  public:
    void push(int);
    ...
};

int main( )
{
    Stack x;
    x.push(2);                  // OK: push( ) is public
    cout << x.top_index;        // Error: cannot access top_index
    return 0;
}
```

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( );
}
```



This Pointer

- Each class member function implicitly contains a pointer of its class type named "this".
- When an object calls the function, this pointer is set to point to the object.
- For example, after compilation, the Stack::push(int x) function in the Stack class will be translated to a unique global function by adding a new argument:

```
void Stack::push(Stack* this, int x)
{
    if (!this->full( ))
        this->data[++(this->top_index)] = x;
}
```

- a.push(x) becomes push(&a, x).

Example: Return an Object by (*this)

```
class Complex                                          /* File: complex.cpp */
{
  private:
    float real; float imag;
  public:
    Complex(float r, float i) { real = r; imag = i; }
    Complex add(const Complex& x)          // Addition of complex numbers
    {
        real += x.real;
        imag += x.imag;
        return *this;
    }
};

int main( )
{
    Complex x(1, 2);
    Complex y(3, 4);
    Complex z = x.add(y);
    return 0;
}
```

File Organization and Separate Compilation

- Suppose you want to write an application using a class called Picture.
- The class developer usually give you 2 files
  - class header file, "picture.h": the class interface
  - class library, "libpicture.a": a binary file consisting of the compiled code of the Picture class' implementation (of constructors, destructor, and other member functions)
- You, the application programmer need to
  - include the Picture class header file in your application programs.
  - link your object files with the Picture class library to produce the final executable.
- In this course, for simplicity, usually we assume that you will be both the class developer and the application programer, and you have the class implementation source files (e.g., picture.cpp).

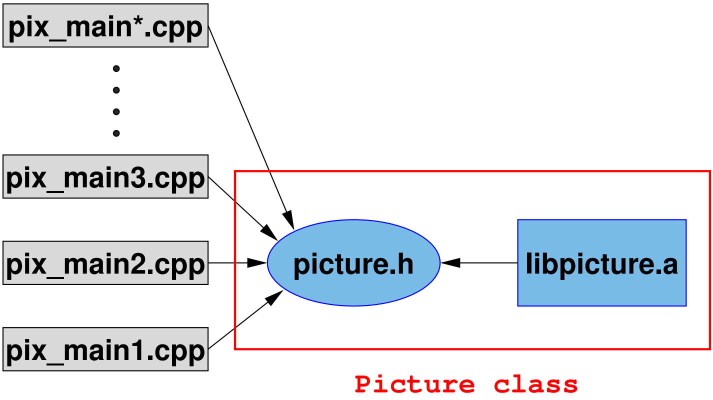
File Organization and Separate Compilation ..

```
/* picture.h */
class Picture
{
    // ...
    Picture* frame(const Picture&);
}
```

```
/* picture.cpp */
#include "picture.h"

Picture* frame(const Picture& x)
{
    ABC_Picture* p
        = new Frame_Picture(x);
    return new Picture(p);
}
// ...
```

```
/* program.cpp */
#include "picture.h"

int main( )
{
    // ...
}
```

Separate Compilation



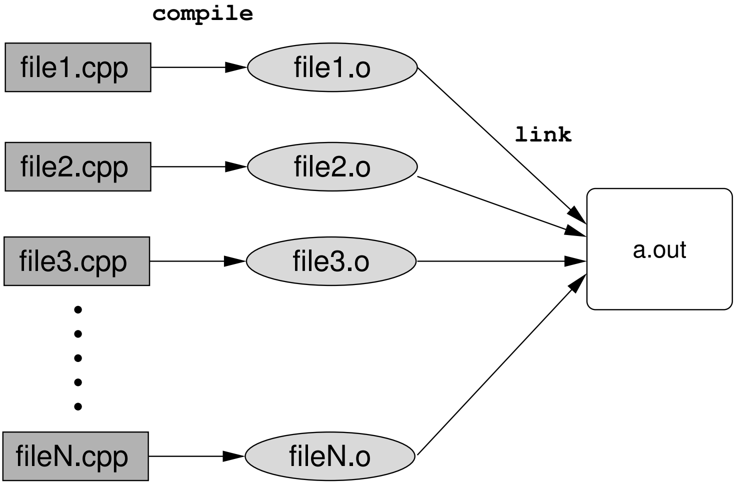
Example: Separate Compilation

- In Linux, compile the program with the GNU C++ compiler as follows:

```
g++ -c program.cpp
g++ -c picture.cpp
g++ -o program program.o picture.o
```

- g++ has many options; man g++ for details.
- The first two lines with "-c" option create the object files "program.o" and "picture.o". They can't run on their own.
- The last line creates the executable program called "program" (with the "-o" option) by linking the object files together.
- Linker: is a program that binds together separately compiled codes.

Linking Object Files



Separate Compilation ..

- If "program.cpp" is changed but "picture.cpp" is not, then the second line is not necessary and you just need:

  ```
  g++ -c program.cpp
  g++ -o program program.o picture.o
  ```

- The separate compilation process can be simplified using "make" on a "Makefile".

- If you don't want the ".o" files, you may compile as follows:

  ```
  g++ -o program program.cpp picture.cpp
  ```

  But then you don't get the object files, "program.o" and "picture.o", but only the executable "program".

Libraries

- If you use any functions declared in the standard C++ header files (iostream, string, etc.), to produce a working executable, the linker needs to include their codes, which can be found in the standard C++ libraries.
- A library is a collection of object files.
- The linker selects object codes from the libraries that contain the definitions for functions used in the program files, and includes them in the executable.
- Some libraries, such as the standard C++ library, are searched automatically by the C++ linker.
- Other libraries have to be specified by the user during the linking process with the '-l" option.

  e.g., To link with the standard math library "libm.a",

  ```
  g++ -o myprog myprog.o -lm
  ```

Preprocessor Directives: #include

- Besides statements allowed in a programming language, some useful program development features are added via directives.
- Directives are handled by a program called preprocessor before the source code is compiled.
- In C++, preprocessor directives begin with the # sign in the very first column.
- The #include directive reads in the contents of the named file.

```
#include <iostream>
#include "myfile.h"
```

- < > are used to include standard header files which are searched at the standard library directories.
- " " are used to include user-defined header files which are searched first at the current directory.
- "g++ -I" may be used to change the search path.

#ifndef, #define, #endif

```
/* program.h */      /* b.h */           /* c.h */
#include "b.h"       #include "a.h"      #include "a.h"
#include "c.h"       #include "d.h"      #include "e.h"
...                  ...                 ...
```

Since #include directives may be nested, the same header file may be included twice!

- multiple processing $\Rightarrow$ waste of time
- re-definition of global variables, constants, classes

Thus, the need of conditional directives

```
#ifndef PICTURE_H
#define PICTURE_H
// object declarations, class definitions, functions
#endif                                          // PICTURE_H
```