04-630 Data Structures and Algorithms for Engineers

Lecture 6: Pointers and Abstract Data Types (ADTs)

Agenda

- Memory layout
- Pointers
- Memory management
- Abstract data types (ADTs)

Pointers and Memory Management: A Primer

Memory Layout

Code(Text)	Contains executable instructions
Data Segment: Static & Globals	Contains static and global variables
Free Store (Heap)	Memory allocated on the Heap must be de-allocated. Allocated dynamically.
Stack	Temporary data: method/function parameters, return address and local variables

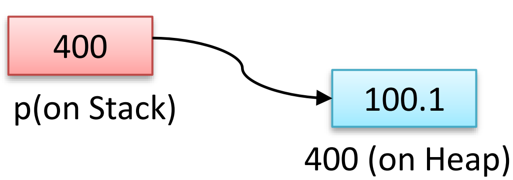
Pointers

- **Recall**: A pointer is a variable that holds the address of another variable.
- Pointers can be allocated statically or dynamically.
- Static allocation:

```
float *p;
float x=100.1;
p=&x; //&: read as address of
cout<<"Memory address pointed to by p:"<<p<<endl;
cout<<"Value at the address pointed to by p:"<<*p<<endl; //*-
dereference
```

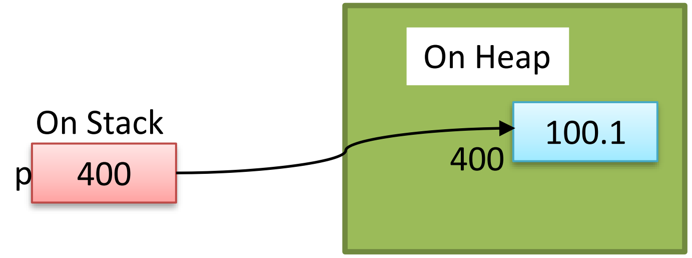
Allocating memory on the Heap

- In C++, we can use the **new** operator in the form:
  - <type> * pointer_var=new <type> e.g.
  - float *p=new float;
  - *p=100.1; //assign a value to the memory the pointer p refers to.
- **new float** allocates memory on the heap.
- The pointer itself is stored on the stack.



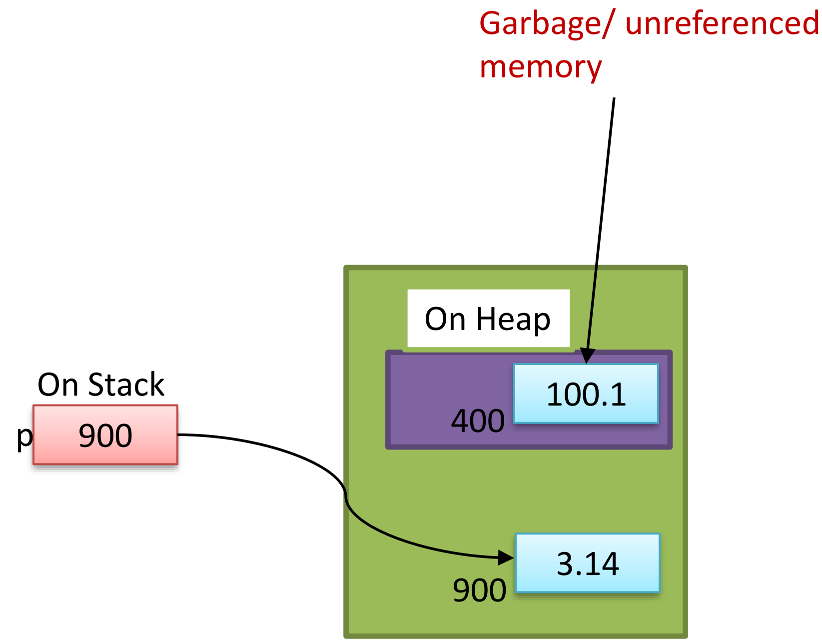
Memory Management

- Consider:
  - `float *p=new float;`
  - `*p=100.1;`



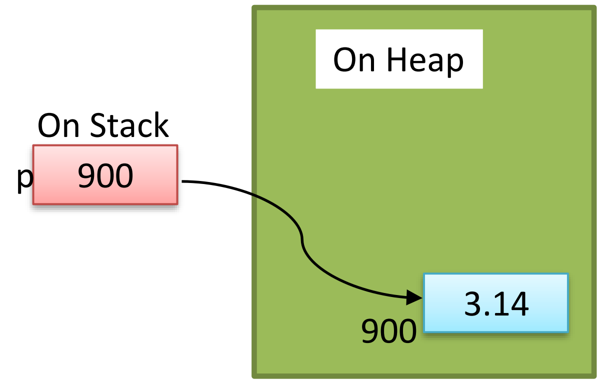
Memory Management

- Consider changing the memory pointed to by p, e.g.:
  - **`p=new float(3.14);`**
- Memory that was holding 100.1 is now unreferenced.
  - Unreferenced memory: garbage.
  - We risk running out of heap space. Why?
  - Program risks crashing.
  - Programmer should explicitly de-allocate memory to avoid garbage.



Memory Management

- Better memory handling, e.g.:
  - `float *p=new float;`
  - `*p=100.1;`
  - `delete p; //frees memory`
  - `p=new float(3.14);`
- delete may lead to a ***dangling pointer***.
  - Dangling pointer: a pointer that is not pointing to a valid object of the relevant type, e.g., after freeing memory.
  - Dangling pointers cause non-deterministic behavior.



Code Sample: allocation and deallocation

```
#include<iostream>
using namespace std;
int main()
{
    float *p=new float;
    *p=100.1;
    cout<<"Address pointed to by p:"<<p<<endl;
    cout<<"Value at address:"<<*p<<endl;
    //delete the memory
    delete p;//step1: free memory
    p=NULL;  //step 2:avoid dangling pointer

    //reassign memory
    p=new float(3.14);
    cout<<"Address pointed to by p:"<<p<<endl;
    cout<<"Value at address:"<<*p<<endl;
    delete p;//free memory
    p=NULL; //avoid dangling pointer
    return 0;
}
```

Memory Management: dangling pointer example

- Consider, e.g.:
  - float *p=new float;
  - *p=100.1;
  - float *p2=p;
  - delete p;
  - p=nullptr; //you can reassign---p=new float(666);

- p2 is pointing to deleted memory.
  - p2 is a dangling pointer.
- Any attempts to use p2 will cause non-deterministic behavior, including program crashes.

Memory management tips

- Use memory checks , e.g. , conditional statements.
- Deallocate/free dynamically allocated memory.
- Avoid dangling pointers.

Abstract Data Types (ADTs)

Abstract Data Types

- Abstract Data Types (ADT): what?
- Information hiding
- Types and typing
- Design goals
- Design practices

Abstract Data Types

- The idea:
  - Specify the complete set of values which a variable of this **type** may assume.
  - Specify completely the set of all possible operations which can be applied to values of this **type.**
  - Do so without reference to the underlying implementation (hence abstract).
  - Achieves **Information hiding.**

Abstract Data Types

- It’s worth noting that object-oriented programming gives us a way of combining (or **encapsulating**) both of these specifications in one logical definition

  – **Class** definition
  (data members and methods, i.e., function members).

  – **Objects** are instantiated classes.

- Actually, object-oriented programming provides much more than this (e.g., inheritance and polymorphism)

Abstract Data Types

Typing and Data Types

- Data types allow programmers to specify what kind of data a variable (or data structure) can store

- Typing is necessary so a computer knows how values in memory will be represented
  - Native types typically include integer, floating point, character, string,...
  - Native data types are built into languages

- ADTs are programmer-defined data types that are created from native types or other ADTs with the express purpose of hiding certain complexities.

Abstract Data Types

An ADT …

- Hides the way information is stored and the details of how the operations do what they do

- Exposes “services” that programmers can use to access, add, delete, manipulate, and transform data

- Are designed for general use, without a particular application or program flow in mind

Abstract Data Types

Example

Native types

```
int MyInteger;
char MyLetter;
float ARealNumber;
```

Programmer defined type

```
#define SIZE 500
struct SomeStructType { /* stack is implemented as */
   char items[SIZE];    /* an array of items      */
   int num;             /* number of items        */
};

typedef struct SomeStructType MyType; /* struct type */

MyType stack;   /* stack is a struct data structure */
```

Abstract Data Types

Example

Native types

```
int MyInteger;
char MyLetter;
float ARealNumber;
```

Programmer defined type

```
#define SIZE 500
struct SomeStructType { /* stack is implemented as */
   char items[SIZE];    /* an array of items       */
   int num;             /* number of items         */
};

typedef struct SomeStructType *MyType; /* pointer type */

MyType stack;  /* variable is a pointer to a stack */
```

Abstract Data Types

ADT Design Goals

- The primary goal in designing an ADT is to hide complexity and implementation details

- Encapsulation is the principle of hiding the way that data is structured, the algorithms used, and providing access to the data and services by way of well-defined interfaces

- We can design systems as a collection of related and interacting capsules that hide various complexities within them

Abstract Data Types

ADT Design Goals

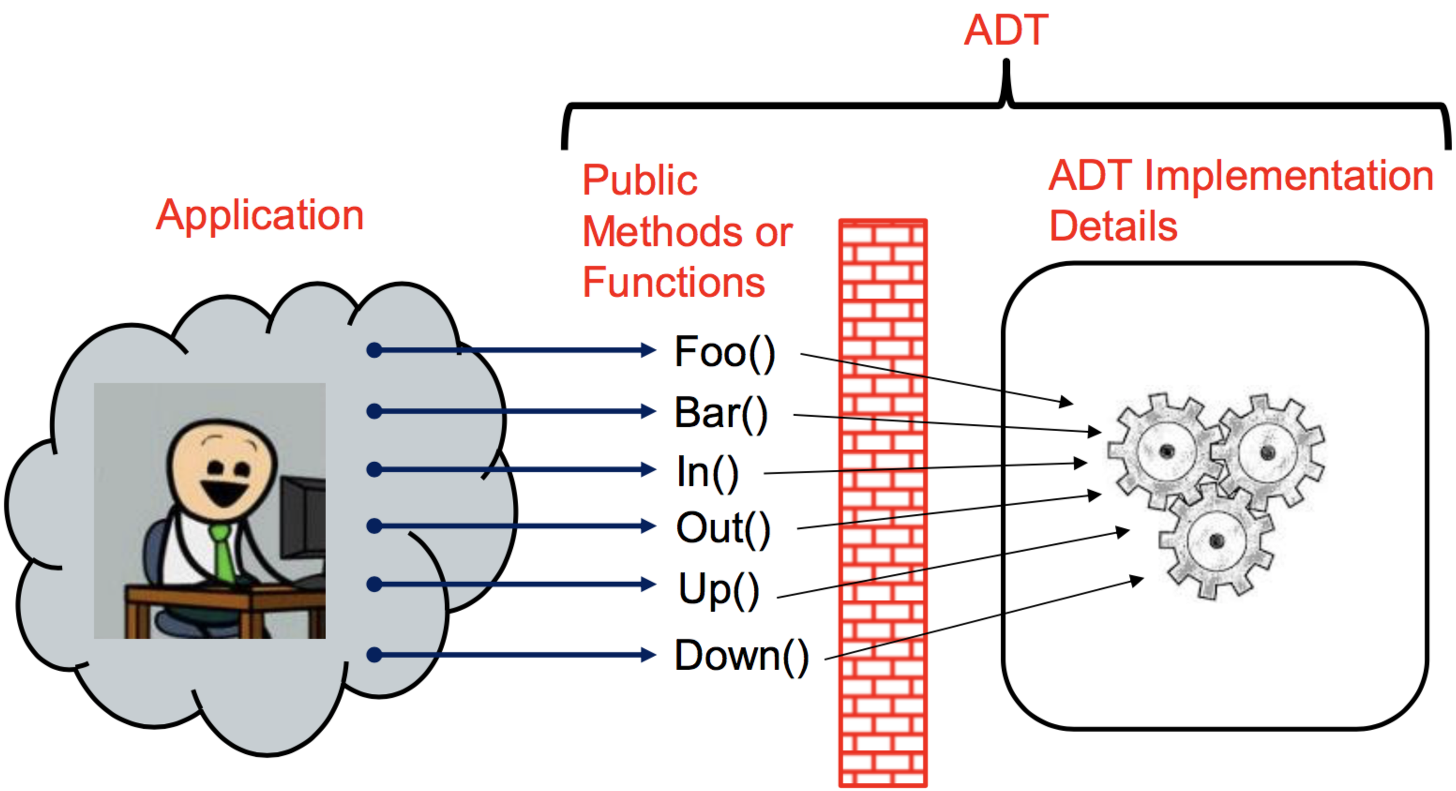
- We must decide
  - What data and operational details do we want to hide?
  - What we have to expose so that (application) programmers can do what they need to do
- We must design
  - The data organization (structure) … & decide how we will allocate resources, store and access data
  - The primitive data types or other ADTs that make up the ADT and the relationships between individual elements of the ADT we are creating
  - The internal and exposed (external) operations are required to operate on the data

Abstract Data Types

ADT Design Goals

- As in all software, in designing an ADT it must be correct:
  - Potentially many applications will depend upon the ADT
  - The data structure or algorithm should work correctly for all possible inputs that might be encountered

Abstract Data Types



Abstract Data Types

Pre- and post-condition checks should be built into ADTs

- protects callers from doing “bad things” and helps prevent logical defects during execution
- checks can be derived from assertions made during algorithm analysis
- The goal in assertion checking is to
  - check the input (preconditions)
  - check for termination (normal and abnormal)
  - check the result (post conditions)

Abstract Data Types

Well designed and developed ADTs can

- Improve usability of services
- Hiding complexity, makes complex operations simple
- Take the underlying implementation understandable
- Facilitate reuse
- Ease maintenance and modifiability

- Poorly designed ADTs can totally undermine these characteristics

Abstract Data Types

Design the ADT before you code

- Decide what the data and operations of the ADT will be before you write the code
- Consider modeling the ADT formally in terms of pre-conditions, post-conditions, invariants...
- ADT operations should do only 1 thing
- Reuse your own operations – never duplicate data or operations
- Think first. Code second.

Abstract Data Types

Decide what you will hide

- Hide as much as possible from the user of the ADT
- Create the most intuitive interfaces possible for programmers
- Reduce inter-module dependencies to the greatest extent possible

Abstract Data Types

- Comment your code – Code is written once and read many times

- Standardize to promote consistency
  - Use coding/comment standards and naming conventions
    use them on internal and external operations and data

- Include headers explaining what the code does
  - Traditionally called “headers” because they are comments at the beginning or “head” of the code body
  - Provides an overview of what the code module does and something about its history

Abstract Data Types

```
/*****************************************************************************
*  Source File: FooBar.c
*  Description: This file contains routines for implementing foobar functions.
*  Author: Gill Bates
*  Initial Production Date: 5/5/07
*  Version: V1.2
*  Calling Convention: FooBar( int FooInput, int BarOutput );
*  Parameter List:
*  int FooInput - integer foo like data
*  int BarOutput - integer bar like data
*  Preconditions: FooInput must be greater than or equal to zero
*  Postconditions: BarOutput will be set to the relevant bar value based on FooInput
*  Functional Abstract: ...
*  ******************************************
*  Revision History
*  12/25/07: Changed FooInput from float to int. Author: Jack Sommers
*  07/04/08: Fixed problem with error message. Author: Jill Smith
```

Abstract Data Types

- Keep your code modules as simple as possible
  - many small, simple operations are often easier to understand than a single complex operation
- Don't be cute and clever with code
  - collapsing 5 lines of code into 1, may yield no advantage at all at execution time
  - complex code is defect prone
  - hard to understand what that 1 line of code does
  - it is better to be sure and correct, than cute and clever and maybe wrong!

Abstract Data Types

- Design error handling from the very beginning

- Good error handling is a result of good correctness and behavioral analysis:
  - error anticipation
  - how the application will react to various types of errors, especially if aborts/halts are possible
  - managing the consequences of errors

Acknowledgement

- Adopted and Adapted from Material by:
- David Vernon: vernon@cmu.edu ; www.vernon.eu