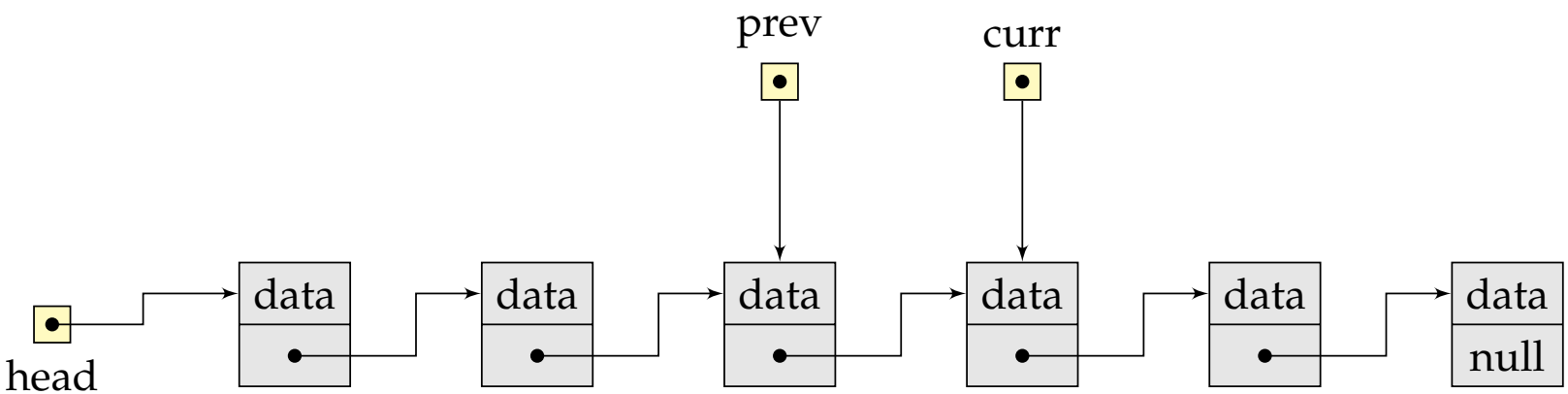
CSCI 2270: Data Structures

Lecture 06: Lists:
Introduction, ArrayList, and C++ Classes

Ashutosh Trivedi



Department of Computer Science
UNIVERSITY OF COLORADO BOULDER

Dynamic Allocation: Quiz

Abstract Data Type: List

Classes in C++

1. What is wrong with this code?

```
// program40.cpp
#include<iostream>
int* foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int *res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << *res << std::endl;
    *res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << *res << std::endl;
    return 0;
  }
}

int* foo(int x) {
  int z = x*x;
  return &z;
}
```

2. What is wrong with this code?

```
// program41.cpp
#include<iostream>
int& foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << res << std::endl;
    res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << res << std::endl;

    return 0;
  }
}

int& foo(int x) {
  int z = x*x;
  return z;
}
```

2. What is wrong with this code?

```
// program41.cpp
#include<iostream>
int& foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << res << std::endl;
    res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << res << std::endl;

    return 0;
  }
}

int& foo(int x) {
  int z = x*x;
  return z;
}
```

Be careful when returning references or pointers.

3. What is wrong with this code?

```
// program42.cpp
#include<iostream>
int* foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int *res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << *res << std::endl;
    *res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << *res << std::endl;
    // Some other computation that uses "res"
    // Some other computation that does not use "res"
    return 0;
  }
}

int* foo(int x) {
  int *z = new int(x*x);
  return z;
}
```

3. Fixed!

```cpp
// program43.cpp
#include<iostream>
int* foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int *res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << *res << std::endl;
    *res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << *res << std::endl;
    // Some other computation that uses "res"
    delete res;
    res = 0;
    // Some other computation that does not use "res"
    return 0;
  }
}

int* foo(int x) {
  int *z = new int(x*x);
  return z;
}
```

4. What is wrong with this code?

```
// program44.cpp
#include<iostream>
int* foo(int  x);
int main(int argc, char* argv[]) {
  if (argc == 2) {
    int* res = foo(atoi(argv[1]));
    // Some other computation that uses "res"
    delete res;
    res = 0;
    // Some other computation that does not use "res"
  }
    return 0;
}
int* foo(int x) {
  int* res = new int[x];
  return res;
}
```

4. Fixed

```cpp
// program45.cpp
#include<iostream>
int* foo(int  x);
int main(int argc, char* argv[]) {
  if (argc == 2) {
    int* res = foo(atoi(argv[1]));
    // Some other computation that uses "res"
    delete[] res;
    res = 0;
    // Some other computation that does not use "res"
  }
    return 0;
}
int* foo(int x) {
  int* res = new int[x];
  return res;
}
```

4. Fixed

```cpp
// program45.cpp
#include<iostream>
int* foo(int  x);
int main(int argc, char* argv[]) {
  if (argc == 2) {
    int* res = foo(atoi(argv[1]));
    // Some other computation that uses "res"
    delete[] res;
    res = 0;
    // Some other computation that does not use "res"
  }
    return 0;
}
int* foo(int x) {
  int* res = new int[x];
  return res;
}
```

Use the same form in corresponding uses of new and delete.

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that has served its purpose.*
5. *Be extremely careful in using references! Use it for speed and memory!*
6. *If the invoked method is not supposed to change the value, use the "const".*

```
void Func3(const int& x);// pass by const reference
```

   This would be used to avoid the overhead of making a copy, but still prevent the data from being changed.
7. *Be mindful that arrays can not be passed by value.*
8. *Be mindful when returning references and pointers.*
9. *Use the same form in corresponding uses of new and delete.*

Dynamic Allocation: Quiz

Abstract Data Type: List

Classes in C++

Abstract Data Types

Abstract data types are an instance of a general principle in software engineering, that combines the following ideas:

1. *Abstraction*. Hiding low-level details with a simpler higher-level idea.
2. *Modularity*. Dividing a system into modules where each module can be separately designed, implemented, and tested.
3. *Encapsulation*. Building walls around the functionality of a module such that bugs in other parts can not damage its integrity, and correctness of the module is its own responsibility.
4. *Information hiding*. Hiding implementation details of a module from the rest of the system, so that those details can be changed without requiring to change the rest of the system.
5. *Separation of concerns*. Making each module responsible for a specific feature (or "concern") rather than distributing responsibilities across multiple modules.

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).
2. *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types along with operations over these types.

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).
2. *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types along with operations over these types.
3. A type is characterized by the operations that you can perform on it:
   - An integer is something that you can add, and multiply;

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).
2. *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types along with operations over these types.
3. A type is characterized by the operations that you can perform on it:
   - An integer is something that you can add, and multiply;
   - A string is something that you can concatenate and find substrings of;

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).
2. *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types along with operations over these types.
3. A type is characterized by the operations that you can perform on it:
   - An integer is something that you can add, and multiply;
   - A string is something that you can concatenate and find substrings of;
   - A boolean is something that you can negate.

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).
2. *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types along with operations over these types.
3. A type is characterized by the operations that you can perform on it:
   - An integer is something that you can add, and multiply;
   - A string is something that you can concatenate and find substrings of;
   - A boolean is something that you can negate.
   - A *list* is linearly-ordered sequence of elements where you can *add*, *remove*, or *get* elements, and compute its size.

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).
2. *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types along with operations over these types.
3. A type is characterized by the operations that you can perform on it:
   - An integer is something that you can add, and multiply;
   - A string is something that you can concatenate and find substrings of;
   - A boolean is something that you can negate.
   - A *list* is linearly-ordered sequence of elements where you can *add*, *remove*, or *get* elements, and compute its size.
   - A *stack* is something that you can *push* a new element on its top, *pop* an element from the top, *check its emptiness*, and so on.

Abstract Data Types

1. Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+`, `-`, `*`, `/`).
2. *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types along with operations over these types.
3. A type is characterized by the operations that you can perform on it:
   - An integer is something that you can add, and multiply;
   - A string is something that you can concatenate and find substrings of;
   - A boolean is something that you can negate.
   - A *list* is linearly-ordered sequence of elements where you can *add*, *remove*, or *get* elements, and compute its size.
   - A *stack* is something that you can *push* a new element on its top, *pop* an element from the top, *check its emptiness*, and so on.
4. In ADT, the specific details of the implementation of the operations are hidden from the user.

Abstract Data Types: Lists

1. A list is a sequence of items where positional order matters, e.g.

$$\langle a_1, a_2, \ldots, a_n \rangle$$

2. Lists are everywhere:
   - lists of students/addresses/patients/appointments,
   - list of processes/files/interrupts
3. Operations required on a list data type:
   - *construct* an empty list
   - *destruct* a list (return the space to the free store.)
   - *insert* an element at a given index on the list.
   - *size()*: return size of the list.
   - *capacity()*: return size of allocated storage capacity.
   - *is_empty()*: check if the list is empty.
4. Let's define the interface and implementation of the List ADT in C++.

Dynamic Allocation: Quiz

Abstract Data Type: List

Classes in C++

Classes: Review

The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types.

—Bjarne Stroustrup

Classes: Review

The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types.

—Bjarne Stroustrup

```
1  // program46.cpp
2  #include<iostream>
3  #include<string>
4
5  /* Class Declaration */
6  class List {
7  private:
8    std::string list[1000];
9    int size;
10   int capacity;
11
12 public:
13   List(); // Constructor
14   ~List(); // Destructor
15   void insert(std::string data);
16   int get_size();
17   int get_capacity();
18   bool is_empty();
19   void pretty_print();
20 };
```

- Access level: public, private (default), and protected.
- Public members can be accessed out of the class (the interface).
- Private members can only be accessed from within the class (data-hiding).

Classes: Review

```
1  /* Class Definition */
2  List::List() {
3    size = 0;
4    capacity = 1000;
5  }
6
7  List::~List() {
8  }
9
10 void List::insert(std::string data) {
11   if (size < capacity) {
12     list[size] = data;
13     size = size + 1;
14   }
15   else {
16     throw "List capacity reached";
17   }
18 }
19 int List::get_size() {
20   return size;
21 }
22
23 int List::get_capacity() {
24   return capacity;
25 }
26
27 bool List::is_empty() {
28   return capacity;
29 }
30
31 void List::pretty_print() {
32   std::cout << "[ ";
33   for (int i = 0; i < size-1; i++) {
34     std::cout << list[i] << ", ";
35   }
36   std::cout <<  list[size-1] <<"]" << std::endl;
37 }
```

Classes: Review

```
1
2  int main(int argc, char* argv[])
3  {
4    List addresses;
5    addresses.insert("Boulder");
6    addresses.insert("Erie");
7    addresses.insert("Louisville");
8    {
9      List names;
10     names.insert("Ashutosh");
11     names.insert("Maciej");
12     names.insert("Shayon");
13
14     names.pretty_print();
15   }
16   addresses.pretty_print();
17
18   return 0;
19 }
```