CENG 112 – Data Structures

Structures and Classes

Mustafa Özuysal
mustafaozuysal@iyte.edu.tr

March 10, 2017

İzmir Institute of Technology

Structures and Compound Data

Compound Data Types

So far we have only looked at primitive data types such as integers, real numbers, characters and arrays composed of these such as integer arrays and strings.

Compound Data Types

So far we have only looked at primitive data types such as integers, real numbers, characters and arrays composed of these such as integer arrays and strings.

In real applications, the data we need to process is usually a composition of fields, which is called a compound data type. Each field might be of a primitive data type or another compound data type.

Compound Data Types

So far we have only looked at primitive data types such as integers, real numbers, characters and arrays composed of these such as integer arrays and strings.

In real applications, the data we need to process is usually a composition of fields, which is called a compound data type. Each field might be of a primitive data type or another compound data type.

Example Names of people can be stored in variables of type string or we can store the name and the surname into two strings and store them in a compund data type.

Compound Data Types

In C/C++, we can create a compund data type using a **`struct`** declaration as follows:

```
struct ContactEntry {
  std::string name;
  std::string surname;
  std::string tel;    // formatted telephone number
                      // such as "+90(555)5555555"
};          // note the obligatory semi-colon at the end
```

Fields can be compound data types

```
struct PhoneNumber {
  std::string country_code;
  std::string area_code;
  std::string tel_number;
};

struct ContactEntry {
  std::string name;
  std::string surname;
  PhoneNumber tel;
};
```

We can statically allocate variables of type `ContactEntry` or arrays of them in a similar way to primitive types. We use the `.` notation to access the fields. Note: We use the simpler `ContactEntry` below with a `tel` field of type `std::string`

```
ContactEntry c;

cout << "Enter contact name: ";
cin >> c.name;
cout << "Enter contact surname: ";
cin >> c.surname;
cout << "Enter contact telephone number: ";
cin >> c.tel;

print_contact(c);
```

We can also pass structures to functions. It more efficient to pass a pointer or reference, otherwise a copy of the whole structure is needed for the variable local to the function.

```
void print_contact(const ContactEntry &contact)
{
        cout << "(" << contact.name << " "
             << contact.surname << ", "
             << contact.tel << ")" << endl;
}
```

Allocating Structures Dynamically

`read_contact()` creates a new structure and returns a pointer to it.

```
ContactEntry *c = read_contact();
print_contact(*c);
delete c;
```

The following DOES NOT work since the statically allocated variable does not exist after the function returns:

```
ContactEntry *read_contact()
{
    ContactEntry c;
    ... code to read from cin ...
    return &c; // This pointer is not valid after function returns.
}
```

Allocating Structures Dynamically

`read_contact()` should dynamically allocate with **new**:

```
ContactEntry *read_contact()
{
        ContactEntry *c = new ContactEntry;
        // c->name is shorthand for (*c).name
        cout << "Enter contact name: ";
        cin >> c->name;
        cout << "Enter contact surname: ";
        cin >> c->surname;
        cout << "Enter contact telephone number: ";
        cin >> c->tel;

        return c;
}

int main(int argc, char** argv)
{
        ContactEntry *c = read_contact();
        print_contact(*c);
        delete c;

        return EXIT_SUCCESS;
}
```

Classes (C++ only)

Classes and Methods

Structures in C can only have data fields. In C++ you can create classes that have both data and function fields. For example, the `ContactEntry` structure had an associated function that print a contact, now we can make it a part of the class definition.

Structures in C can only have data fields. In C++ you can create classes that have both data and function fields. For example, the `ContactEntry` structure had an associated function that print a contact, now we can make it a part of the class definition.

More importantly, C++ let's you hide data and function members so that you can change them later. This is called encapsulation and will be a major theme in CENG211. We will simply put the user visible data and functions under the `public` interface and the rest in the `private` section.

ContactItem class

We can put the implementation of short function into the class declaration

```
class ContactItem {
public:
        std::string name()    { return m_name; }
        std::string surname() { return m_surname; }
        std::string tel()     { return m_tel; }

        void print();
        void read();
private:
        std::string m_name;
        std::string m_surname;
        std::string m_tel;
};
```

`ContactItem` class

We put the implementation of longer functions in a seperate source file

```cpp
void ContactItem::read()
{
        cout << "Enter contact name: ";
        cin >> m_name;
        cout << "Enter contact surname: ";
        cin >> m_surname;
        cout << "Enter contact telephone number: ";
        cin >> m_tel;
}

void ContactItem::print()
{
        cout << "(" << m_name << " "
             << m_surname << ", "
             << m_tel << ")" << endl;
}
```

ContactItem class

Using this class is simpler

```
int main(int argc, char** argv)
{
        ContactItem c;
        c.read();
        c.print();
        return EXIT_SUCCESS;
}
```

Resizable Array Class

Capacity and Size

- We will implement an `std::vector` like class that will store and manage arrays of arbitrary types.

Capacity and Size

- We will implement an `std::vector` like class that will store and manage arrays of arbitrary types.
- We will start by writing an `IntArray` class that manages an dynamically allocated integer array stored as `int *`.

Capacity and Size

- We will implement an `std::vector` like class that will store and manage arrays of arbitrary types.
- We will start by writing an `IntArray` class that manages an dynamically allocated integer array stored as `int *`.
- `IntArray`s have two important properties: capacity and size

Capacity and Size

- We will implement an `std::vector` like class that will store and manage arrays of arbitrary types.
- We will start by writing an `IntArray` class that manages an dynamically allocated integer array stored as `int *`.
- `IntArray`s have two important properties: capacity and size
- Capacity is the length of the dynamically allocated array

Capacity and Size

- We will implement an `std::vector` like class that will store and manage arrays of arbitrary types.
- We will start by writing an `IntArray` class that manages an dynamically allocated integer array stored as `int *`.
- `IntArray`s have two important properties: capacity and size
- Capacity is the length of the dynamically allocated array
- Size is the current number of integers stored in the `IntArray`

Capacity and Size

- We will implement an `std::vector` like class that will store and manage arrays of arbitrary types.
- We will start by writing an `IntArray` class that manages an dynamically allocated integer array stored as `int *`.
- `IntArray`s have two important properties: capacity and size
- Capacity is the length of the dynamically allocated array
- Size is the current number of integers stored in the `IntArray`
- Size will always be less than or equal to capacity.

Constructor/Destructor

- One important question is when to allocate and deallocate the array.

Constructor/Destructor

- One important question is when to allocate and deallocate the array.
- The array should be allocated when the `IntArray` is created and deallocated when the `IntArray` is destroyed.

Constructor/Destructor

- One important question is when to allocate and deallocate the array.
- The array should be allocated when the `IntArray` is created and deallocated when the `IntArray` is destroyed.
- C++ allows us to define two special functions that will run at object creation and destruction: constructor and destructor.

Constructor/Destructor

- One important question is when to allocate and deallocate the array.
- The array should be allocated when the `IntArray` is created and deallocated when the `IntArray` is destroyed.
- C++ allows us to define two special functions that will run at object creation and destruction: constructor and destructor.
- Constructor has the same name as the class, in our case `IntArray::IntArray`.

Constructor/Destructor

- One important question is when to allocate and deallocate the array.
- The array should be allocated when the `IntArray` is created and deallocated when the `IntArray` is destroyed.
- C++ allows us to define two special functions that will run at object creation and destruction: constructor and destructor.
- Constructor has the same name as the class, in our case `IntArray::IntArray`.
- Destructor has the same name as the class but prefixed by ~, `IntArray::~IntArray`.

Constructor/Destructor

- One important question is when to allocate and deallocate the array.
- The array should be allocated when the `IntArray` is created and deallocated when the `IntArray` is destroyed.
- C++ allows us to define two special functions that will run at object creation and destruction: constructor and destructor.
- Constructor has the same name as the class, in our case `IntArray::IntArray`.
- Destructor has the same name as the class but prefixed by ~, `IntArray::~IntArray`.
- Both of them have an empty return type.

Public Section

Let's start with the public functions that users of this data type will call

```
class IntArray {
  public:
  IntArray();  // constructor
  ~IntArray(); // destructor

  int size();
  int capacity();
  void insert(int item);
  int  item_at(int index);
  void clear(); // removes all items
};
```

Private Section

The private section will store the array and information on size and capacity

```
class IntArray {
  public:
  ..........

  private:
  int *m_items;
  int m_size;
  int m_capacity;
};
```

Implementing Constructor and Destructor

We need to allocate and deallocate the array

```
IntArray::IntArray()
{
        const int INITIAL_CAPACITY = 8;

        m_items = new int[INITIAL_CAPACITY];
        m_capacity = INITIAL_CAPACITY;
        m_size = 0;
}

IntArray::~IntArray()
{
        delete [] m_items;
}
```

Implementing Helper Functions

```cpp
int IntArray::size()
{
        return m_size;
}

int IntArray::capacity()
{
        return m_capacity;
}

int  IntArray::item_at(int index)
{
        return m_items[index];
}

void IntArray::clear()
{
        m_size = 0;
}
```

Implementing Insertion

Insertion needs to check capacity and grow the array if necessary

```
void IntArray::insert(int item)
{
        if (m_size >= m_capacity) {
                int new_cap = 2*m_capacity;
                int *new_items = new int[new_cap];
                for (int i = 0; i < m_size; ++i)
                        new_items[i] = m_items[i];
                delete [] m_items;
                m_items = new_items;
                m_capacity = new_cap;
        }

        m_items[m_size] = item;
        m_size++;
}
```

IntArray In Action

```
IntArray ia;
cout << "Capacity/Size at creation = " << ia.capacity()
     << "/" << ia.size() << endl;

for(int i = 0; i < 10; ++i)
        ia.insert(i);
cout << "Capacity/Size after insertions = " << ia.capacity()
     << "/" << ia.size() << endl;
cout << "Items: ";
for(int i = 0; i < ia.size(); ++i)
        cout << ia.item_at(i) << " ";
cout << endl;

ia.clear();
cout << "Capacity/Size after clearing = " << ia.capacity()
     << "/" << ia.size() << endl;
```

IntArray In Action

```cpp
IntArray ia;
cout << "Capacity/Size at creation = " << ia.capacity()
     << "/" << ia.size() << endl;

for(int i = 0; i < 10; ++i)
        ia.insert(i);
cout << "Capacity/Size after insertions = " << ia.capacity()
     << "/" << ia.size() << endl;
cout << "Items: ";
for(int i = 0; i < ia.size(); ++i)
        cout << ia.item_at(i) << " ";
cout << endl;

ia.clear();
cout << "Capacity/Size after clearing = " << ia.capacity()
     << "/" << ia.size() << endl;
```

```
Capacity/Size at creation = 8/0
Capacity/Size after insertions = 16/10
Items: 0 1 2 3 4 5 6 7 8 9
Capacity/Size after clearing = 16/0
```

Generic Resizable Array

- `IntArray` class is good but it can only manage integer arrays

Generic Resizable Array

- `IntArray` class is good but it can only manage integer arrays
- Do we need to write `FloatArray`, `StringArray`, `ContactArray`, . . . ?

Generic Resizable Array

- `IntArray` class is good but it can only manage integer arrays
- Do we need to write `FloatArray`, `StringArray`, `ContactArray`, . . . ?
- No, we just need a way to replace `int` in `IntArray` declaration and definition with any possible type.

Generic Resizable Array

- `IntArray` class is good but it can only manage integer arrays
- Do we need to write `FloatArray`, `StringArray`, `ContactArray`, . . . ?
- No, we just need a way to replace `int` in `IntArray` declaration and definition with any possible type.
- In C++, we do that with templates.

Generic Resizable Array

- `IntArray` class is good but it can only manage integer arrays
- Do we need to write `FloatArray`, `StringArray`, `ContactArray`, . . . ?
- No, we just need a way to replace `int` in `IntArray` declaration and definition with any possible type.
- In C++, we do that with templates.
- We just need to write `template <typename T>` at the beginning and replace `int` by `T` in a new class `Array`.

Generic Resizable Array

- `IntArray` class is good but it can only manage integer arrays
- Do we need to write `FloatArray`, `StringArray`, `ContactArray`, . . . ?
- No, we just need a way to replace `int` in `IntArray` declaration and definition with any possible type.
- In C++, we do that with templates.
- We just need to write `template <typename T>` at the beginning and replace `int` by `T` in a new class `Array`.
- We will also put our classes into the `ceng112` namespace to avoid name clashes.

Array Usage

We want to use the new Array class like this, note the way we index

```
Array<int> ia;
cout << "Capacity/Size at creation = " << ia.capacity()
     << "/" << ia.size() << endl;

for(int i = 0; i < 10; ++i)
        ia.insert(i);
cout << "Capacity/Size after insertions = " << ia.capacity()
     << "/" << ia.size() << endl;
cout << "Items: ";
for(size_t i = 0; i < ia.size(); ++i)
        cout << ia[i] << " ";
cout << endl;

ia.clear();
cout << "Capacity/Size after clearing = " << ia.capacity()
     << "/" << ia.size() << endl;
```

Array Usage

We should be able to work with **`float`** arrays, it would be nice if wee could directly print the arrays

```
Array<float> fa;
cout << fa << endl;

for(float i = 0.1f; i < 10.1f; ++i)
        fa.insert(i);
cout << fa << endl;

fa.clear();
cout << fa << endl;
```

```
[(C/S=0/0)]
[0.1 1.1 2.1 3.1 4.1 5.1 6.1 7.1 8.1 9.1 (C/S=16/10)]
[(C/S=16/0)]
```

Array Declaration

```
template <typename T>
class Array {
public:
        Array() { m_items = 0; m_size = 0; m_capacity = 0; }
        ~Array() { delete [] m_items; }

        size_t size() const { return m_size; }
        size_t capacity() const { return m_capacity; }
        void clear() { resize(0); }

        void resize(int new_size);
        void insert(T item);

        T& operator[](size_t index) { return m_items[index]; }
        const T& operator[](size_t index) const { return m_items[index
private:
        T *m_items;
        size_t m_size;
        size_t m_capacity;
};
```

Array `insert` Implementation

Important Note: This also goes into the header since the compiler needs to see the definition to create versions (`Array<int>`, `Array<float>`, ...) as necessary.

```
template <typename T>
void Array<T>::insert(T new_item)
{
        resize(m_size+1);
        m_items[m_size-1] = new_item;
}
```

Array `resize` Implementation

Important Note: This also goes into the header

```
template <typename T>
void Array<T>::resize(int new_size)
{
        if (new_size > m_capacity) {
                size_t new_cap = (m_capacity > 0) ? m_capacity : 1;
                while (new_size > new_cap)
                        new_cap *= 2;
                T * new_items = new T[new_cap];
                for (size_t i = 0; i < m_size; ++i)
                        new_items[i] = m_items[i];
                delete [] m_items;
                m_items = new_items;
                m_capacity = new_cap;
        }

        m_size = new_size;
}
```

Printing for Your Classes

We can use **`cout <<`** to print our classes if we implement a function taking an **`std::ostream &`** and a constant reference to an object of our class. **Important Note:** This also goes into the header

```cpp
template <typename T>
std::ostream& operator<<(std::ostream& os, const Array<T>& arr)
{
        os << "[";
        for (size_t i = 0; i < arr.size(); ++i)
                os << arr[i] << " ";
        os << "(C/S=" << arr.capacity() << "/" << arr.size();
        os << ")]";
        return os;
}
```