Abstract Data Types

- Often times we want to represent abstract things (beyond an integer, character, or double)
  - Examples:
    - A pixel (with R,G,B value)
    - A file (filename, start location on disk, current pointer to location in the file to read/write)
    - A student (name, ID, major)
- Often these abstract types can be represented as a collection of integers, character arrays, etc.
- Structs and later Classes in C++ allow us to create collections of heterogeneous variables
  - Can reference the collection with a single name (pixelA, student1)
  - Can access individual components (pixelA.red, student1.id)
Struct: Definitions and Instances (Declarations)

- **A struct must first be defined**
  - The declaration is a blueprint that indicates what any instance should look like
  - Identifies the overall name of the struct and its individual component types and names
  - The declaration does not actually create a variable
  - Usually appears outside any function
- **Then any number of instances can be created/declared in your code**
  - Instances are actual objects created from the definition (blueprint)
  - Declared like other variables

```cpp
#include<iostream>
using namespace std;

// struct definition
struct pixel {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};

int main() {
    int i, j;
    // instantiations
    struct pixel pixela;
    struct pixel image[256][256];
    // make pixela red
    pixela.red = 255;
    pixela.blue = pixela.green = 0;
    // make a green image
    for(i=0; i < 256; i++){
        for(j=0; j < 256; j++){
            image[i][j].green = 255;
            image[i][j].blue = 0;
            image[i][j].red = 0;
        }
    }
    return 0;
}
```
Memory View of Struct’s

- Each instantiation allocates memory for all the members/components of the struct

```cpp
#include <iostream>

using namespace std;

struct pixel {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};

int main() {
    int i, j;
    // instantiations
    struct pixel pixela;
    struct pixel image[256][256];
    ...
    return 0;
}
```
• Structs can contain arrays or even other structs

```cpp
#include<iostream>

using namespace std;

struct student {
    char name[80];
    int id;
    int major;
};

int main( )
{
    int i,j;
    // instantiations
    struct student s1;
... 
    return 0;
}
```
Accessing struct members

- Each variable in a struct definition is called a ‘member’ of the struct.
- When declaring an instance/variable of a struct, we give the entire struct a name, but the individual members are identified with the name provided in the definition.
- We can access that member in an instance of the struct by using the . (dot) operator.

```cpp
#include<iostream>

using namespace std;

struct student {
    char name[80];
    int id;
    int major;
};

int main()
{
    struct student my_student;
    cout <<“ Enter your name” <<endl;
    cin >> my_student.name;
    cout <<“ Enter your id” <<endl;
    cin >>my_student.id;
    cout <<“ Enter your major(1/2)” <<endl;
    cin >>my_student.major;
    if(my_student.major == 2)
        cout << “You like HW” << endl;
    else
        cout << “You like SW” << endl;
...
    return 0;
}
```
Struct assignment

- Will perform an element by element copy of the source struct to the destination struct

```cpp
#include<iostream>

using namespace std;

struct student {
    char name[80];
    int id;
    int major;
};

int main() {
    struct student s1,s2;
    s1.name = "Bill";
    s1.id = 5;
    s1.major = 1;
    s2 = s1;
    return 0;
}
```
Pointers to structs

• We can declare pointers to structs just as any other variable

```cpp
#include<iostream>

using namespace std;

struct student {
    char name[80];
    int id;
    int major;
};

int main()
{
    struct student s1,*stu_ptr;
    s1.name = "Bill";
    s1.id = 5;
    s1.major = 1;
    stu_ptr = &s1;
    return 0;
}
```
Accessing members from a Pointer

- Can dereference the pointer first then use the dot operator

```cpp
#include<iostream>

using namespace std;

struct student {
    char name[80];
    int id;
    int major;
};

int main()
{
    struct student s1, *stu_ptr;
    s1.name = "Bill";
    s1.id = 5;
    s1.major = 1;
    stu_ptr = &s1;
    (*stu_ptr).id = 4;
    (*stu_ptr).name = "Tom";

    return 0;
}
```
Arrow (->) operator

- Shortcut
  - (*struct_ptr).member equivalent to struct_ptr->member

```cpp
#include <iostream>

using namespace std;

struct student {
  char name[80];
  int id;
  int major;
};

int main() {
  struct student s1, *stu_ptr;
  s1.name = "Bill";
  s1.id = 5;
  s1.major = 1;
  stu_ptr = &s1;
  stu_ptr->id = 4;
  stu_ptr->name = "Tom";

  return 0;
}
```
Passing Structs as Arguments

- In C++, arguments must be a single value [i.e. a single data object / can’t pass an entire array of data, instead pass a pointer]
- Structs are the exception...you can pass an entire struct ‘by value’
  - Will make a copy of the struct and pass it to the function
- Of course, you can always pass a pointer [especially for big structs since pass by value means making a copy of a large struct]
Returning Structs

- Can only return a single struct from a function [i.e. not an array of structs]
  - For an array you pass it in by pointer and modify the original array
- Will return a copy of the struct indicated

```cpp
#include<iostream>
#define CS 1
#define CECS 2
using namespace std;

struct Point {
    int x;
    int y;
};

void print_point(struct Point *myp) {
    cout << "(x,y)=" << myp->x << "," << myp->y;
    cout << endl;
}

struct Point make_point() {
    struct Point temp;
    temp.x = 3; temp.y = -1;
    return temp;
}

int main() {
    struct Point p1;
    p1 = make_point(); print_point(&p1);
    return 0;
}
```
Object-Oriented Programming

• Model the application/software as a set of objects that interact with each other
• Objects fuse data (i.e. variables) and functions (a.k.a methods) that operate on that data into one item (i.e. object)
  – Like structs but now with associated functions/methods
• Objects become the primary method of encapsulation and abstraction
  – Encapsulation
    • Hiding of data and implementation details (i.e. make software modular)
    • Only expose a well-defined interface to anyone wanting to use our object
  – Abstraction
    • How we decompose the problem and think about our design rather than the actual code
Objects

• Objects contain:
  – Data members
    • Data needed to model the object and track its state/operation (just like structs)
  – Methods/Functions
    • Code that operates on the object, modifies it, etc.

• Example: Deck of cards
  – Data members:
    • Array of 52 entries (one for each card) indicating their ordering
    • Top index
  – Methods/Functions
    • Shuffle(), Cut(), Get_top_card()
C++ Classes

- Programming construct used to define objects, their data members, and methods/functions
- Similar idea to structs
- Steps:
  - Define the class’ data members and method prototypes (usually in a separate header file)
  - Write the methods (usually in a separate .cpp file)
  - Instantiate/Declare object variables and use them by calling their methods
- Terminology:
  - **Class** = Definition/Blueprint of an object
  - **Object** = Instance of the class, actual allocation of memory, variable, etc.

```cpp
class Deck {
public:
    Deck();   // Constructor
    ~Deck();  // Destructor
    void shuffle();
    void cut();
    int get_top_card();
private:
    int cards[52];
    int top_index;
};
```

```cpp
#include <iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
    Deck d;
    int hand[5];
    d.shuffle();
    d.cut();
    for(int i=0; i < 5; i++)
    {
        hand[i] = d.get_top_card();
    }
}
```
Class Definition

- **class name { ... };**
- Each function or data member can be classified as **public, private, or protected**
  - These classifications support encapsulation by allowing data/method members to be inaccessible to code that is not a part of the class (i.e. only accessible from within a public class method)
  - **Private**: Can call or access only by methods/functions that are part of that class
  - **Public**: Can call or access by any other code
  - **Protected**: More on this later
- Everything private by default so you must use “public:” to make things visible
- Make the interface public and the guts/inner-workings private

```cpp
#include <iostream>
#include "deck.h"
// Code for each prototyped method

int main(int argc, char *argv[]) {
    Deck d;
    int hand[5];
    d.shuffle();
    d.cut();
    d.cards[0] = ACE; // won’t compile
    d.top_index = 5; // won’t compile
}
```
Constructors / Destructors

- Constructor is a function of the same name as the class itself
  - It is called automatically when the object is created
  - Use to initialize your object to desired state
  - Returns nothing

- Destructor is a function of the same name as class itself with a ‘~’ in front
  - Called automatically when object goes out of scope
  - Use to delete any memory allocated by the object
  - Returns nothing

```cpp
class Deck {
public:
    Deck();    // Constructor
    ~Deck();   // Destructor
    ...
};
```

```cpp
#include<iostream>
#include "deck.h"

Deck::Deck() {
    top_index = 0;
    for(int i=0; i < 52; i++) {
        cards[i] = i;
    }
}

Deck::~Deck() {
}
```

```cpp
#include<iostream>
#include "deck.h"

int main() {
    Deck d;    // Deck() is called
    ...
    return 1;
    // ~Deck() is called since // function is done
}
```
UML (Unified Modeling Language)

- Shows class definitions in a language-agnostic way
- Shows class hierarchy (inheritance, etc.)
- Each class shown in one box with 3 sections
  - Class Name, Member functions, then Data members
  - Precede function/data member with:
    + (public), - (private), # (protected)
  - Functions show name with arguments : return type
  - Data members show name : type

```cpp
class Deck {
public:
    Deck();       // Constructor
    ~Deck();      // Destructor
    void shuffle();
    void cut();
    int get_top_card();

private:
    int cards[52];
    int top_index;
};
```
When writing member functions, the compiler somehow needs to know that the function is a member of a particular class, thus we must ‘scope’ our functions.

Include the name of the class followed by ‘::’ just before name of function.

This allows the compiler to check access to private/public variables.

- Without the scope operator [i.e. void shuffle() rather than void Deck::shuffle()] the compiler would think that the function is some outside function (not a member of Deck) and thus generate an error when it tried to modify the cards array.
Member functions are called by preceding their name with the specific object that it should operate on.

d1.shuffle() indicates the code of shuffle() should be operating implicitly on d1’s data member vs. d2 or any other Deck object.
Calling Member Functions

- Within a member function we can just call other member functions directly.

```cpp
#include <iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
  Deck d1, d2;
  int hand[5];
  d1.shuffle();
  ...
}
```

```cpp
#include <iostream>
#include "deck.h"

void Deck::shuffle()
{
  cut(); // calls cut()
  // for this object
  for(i=0; i < 52; i++){
    int r = rand() % (52-i);
    int temp = cards[r];
    cards[r] = cards[i];
    cards[i] = temp;
  }
}

void Deck::cut()
{
  // swap 1st half of deck w/ 2nd
}
```

D1’s data will be modified (shuffled and cut)  
D1 is implicitly passed to shuffle()  
D2

Since shuffle was implicitly working on d1’s data, d1 is again implicitly passed to cut()
Strings

- **Must:**
  - `#include <string>`
  - `using namespace std;`

- **Initializations / Assignment**
  - Use *initialization constructor*
  - Use `=‘ operator`
  - Can reassign and all memory allocation will be handled

- **Redefines operators:**
  - `+` (concatenate / append)
  - `+=` (append)
  - `==, !=, >, <, <=, >=` (comparison)
  - `[]` (access individual character)

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    int len;
    string s1("Timothy D.");
    string s2;
    s2 = "Christopher";
    s2 = "Allison";
    cout << s1 << " & " << s2 << endl;
    cout << "My name is " << s1 << endl;
    s2 = s2 + "!!!";
    cout << s2 << endl;
    string s3 = s1;
    if (s1 == s3) {
        cout << s1 << " same as " << s3;
        cout << endl;
    }
    cout << "First letter is “ << s1[0];
    cout << endl;
}
```

**Output:**
Timothy D. & Allison
My name is Timothy D.
Allison!!!
Timothy D. same as Timothy D.
First letter is T
More String Examples

- **Size/Length of string**
- **Find a substring**
  - Searches for occurrence of a substring
  - Returns either the index where the substring starts or `std::npos`
  - `std::npos` is a constant meaning ‘just beyond the end of the string’...it’s a way of saying ‘Not found’
- **Get a substring**
  - Pass it the start character and the number of characters to copy
  - Returns a new string
- **Others**: replace, `rfind`, etc.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s1("abc def");
    cout << "Len of s1: " << s1.size() << endl;

    if(s1.find("bc d") != string::npos)
        cout << "Found bc_d starting at pos=":
        cout << s1.find("bc_d") << endl;

    found = s1.find("def");
    if( found != string::npos){
        string s2 = s1.substr(found,3)
        cout << s2 << endl;
    }
}
```

**Output:**
```
Len of s1: 7
Found bc_d starting at pos=1
def
```
Public / Private

- Recall unlike structs where other code can access member variables, other code (non-member functions of the class) **cannot** access private class members directly.

```cpp
class Student {
public:
    Student();  // Constructor 1
    Student(string name, int id, double gpa);
    ~Student();  // Destructor
...
private:
    string _name;
    int _id;
    double _gpa;
};
```

```cpp
#include<iostream>
#include <string>

int main()
{
    Student s1;  string myname;
    cin >> myname;
    s1._name = myname;  //compile error
    ...
}
```
Accessor / Mutator Methods

- Define public "get" (accessor) and "set" (mutator) functions to let other code access desired private data members.

```cpp
#include<iostream>
#include <string>

int main()
{
    Student s1;  string myname;
    cin >> myname;
    s1.set_name(mynname);

    string another_name;
    another_name = s1.get_name();
    ...
}
```

```cpp
class Student {
public:
    Student();   // Constructor 1
    Student(string name, int id, double gpa);  // Constructor 2
    ~Student();  // Destructor
    string get_name();
    int get_id();
    double get_gpa();
    void set_name(string s);
    void set_id(int i);
    void set_gpa(double g);

private:
    string _name;
    int _id;
    double _gpa;
};
```

```cpp
string Student::get_name()
{
    return _name;
}
int Student:: get_id()
{
    return _id;
}
void Student:: set_name(string s)
{
    _name = s;
}
void Student:: set_gpa(double g)
{
    _gpa = g;
}
```
Multiple Constructors

• Can have multiple constructors with different argument lists

```cpp
#include <iostream>
#include <string>

int main() {
    Student s1; // calls Constructor 1
    string myname;
    cin >> myname;
    s1.set_name(mynname);
    s1.set_id(214952);
    s1.set_gpa(3.67);
    Student s2(mynname, 32421, 4.0); // calls Constructor 2
}
```

```cpp
class Student {
public:
    Student(); // Constructor 1
    Student(string name, int id, double gpa); // Constructor 2
    ~Student(); // Destructor
    string get_name();
    int get_id();
    double get_gpa();
    void set_name(string name);
    void set_id(int id);
    void set_gpa(double gpa);
private:
    string _name;
    int _id;
    double _gpa;
};
```