CS 103 Unit 9 – Objects, Structs, and Strings

Mark Redekopp
OBJECTS
Motivation for Objects

• Would you rather carry all your luggage items piecemeal or pack everything in one suitcase?

```c
void process(char name[], int id, int major) {
}
```

```c
void process(Student s1) {
}
```
Objects

- Often times we want to represent higher level concepts, objects, or things (beyond an integer, character, or double)
  - Examples: a pixel, a circle, or a student
- Often these "objects" can be represented as a collection of integers, character arrays/strings, etc.
  - A pixel (with R,G,B value)
  - A circle (center\_x, center\_y, radius)
  - A student (name, ID, major)
- Objects (realized as 'structs' in C and later 'classes' in C++) allow us to aggregate different type variables together to represent a single larger 'thing' as well as supporting operations on that 'thing'
  - Can reference the collection with a single name (myCircle, student1)
  - Can access individual components (myCircle.radius, student1.id)
Object-Oriented Approach

- 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)
- Objects replace global-level functions as the primary method of **encapsulation** and **abstraction**
  - **Encapsulation**: Code + data together with controlled access
    - Group data and code that operates on that data together into one unit
    - Only expose a well-defined interface to control misuse of the code by other programmers
  - **Abstraction**
    - Hiding of data and implementation details
    - How we decompose the problem and think about our design at a higher level rather than considering everything at the lower level
Object-Oriented Programming

• 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()
Types and Instances

• A 'type' indicates how much memory will be required, what the bits mean (i.e. data vs. address), and what operations can be performed
  – int = 32-bits representing only integer values and supporting +, -, *, /, =, ==, <, >, etc.
  – char* = 32-bit representing an address and supporting * (dereference), &+, - (but not multiply and divide)
  – Types are like blueprints for what & how to make a particular 'thing'

• A variable or object is an actual instantiation (allocation of memory) for one of these types
  – int x, double z, char *str;
C++ Objects: Structs vs. Classes

• **Structs** (originated in the C language) are the predecessors of **classes** (C++ language)
  – Though **structs** are still valid in C++

• **Classes** form the basis of ‘object-oriented’ programming in the C++ language

• Both are simply a way of *grouping* related **data** together and related **operations** (functions or methods) to model some 'object'
  – The majority of the following discussion applies both to **structs** and **classes** equally so pay attention now to make next lecture easier.
Starting with data...

STRUCTS
Definitions and Instances (Declarations)

- Objects must first be defined/declared (as a 'struct' or 'class')
  - The declaration is a blue print 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/instantiated 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;
};
// 'pixel' is now a type
// just like 'int' is a type
int main(int argc, char *argv[]) {
    int i,j;
    // instantiations
    pixel pixela;
    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;
}
```
Membership Operator (.)

- Each variable (and function) in an object definition is called a member of the object (i.e. struct or class)
- When declaring an instance/variable of an object, we give the entire object a name, but the individual members are identified with the member names provided in the definition
- We use the . (dot/membership) operator to access that member in an instance of the object
  - Supply the name used in the definition above so that code is in the form:
  
    instance_name.member_name

```cpp
#include <iostream>
using namespace std;
enum {CSCI=1, CECS=2};
struct student {
    char name[80];
    int id;
    int major;
};

int main(int argc, char *argv[]) {
    int i,j;
    // instantiations
    student my_student;
    // setting values
    strncpy(my_student.name,"Tom Trojan",80);
    my_student.id = 1682942;
    my_student.major = CSCI;
    ...
    return 0;
}
```
Memory View of Objects

- Each instantiation allocates memory for all the members/components of the object (struct or class)

```cpp
#include<iostream>

using namespace std;

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

int main(int argc, char *argv[]) {
    int i,j;
    // instantiations
    pixel pixela;
    pixel image[256][256];
    ...
    return 0;
}
Memory View of Objects

- Objects can have data members that are arrays or even other objects

```cpp
#include<iostream>

using namespace std;

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

int main(int argc, char *argv[])
{
    int i,j;
    // instantiations
    student s1;
    ...
    return 0;
}
```
Assignment semantics and pointers to objects

IMPORTANT NOTES ABOUT OBJECTS
Object assignment

• Consider the following initialization of s1

```cpp
#include<iostream>
using namespace std;
enum {CSCI=1, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[])
{
    student s1, s2;
    strncpy(s1.name, "Jill", 80);
    s1.id = 5; s1.major = CECS;
}```
# Object assignment

- Assigning one object to another will perform an member-by-member copy of the entire source object to the destination object.

```cpp
#include <iostream>
using namespace std;
enum {CSCI=1, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[]) {
    student s1, s2;
    strncpy(s1.name, "Jill", 80);
    s1.id = 5; s1.major = CECS;
    s2 = s1;
    return 0;
}
```
Pointers to Objects

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

```cpp
#include<iostream>
using namespace std;
enum {CSCI=1, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[]) {
    student s1, *stu_ptr;
    strncpy(s1.name,"Jill",80);
    s1.id = 5; s1.major = CECS;
    stu_ptr = &s1;
    return 0;
}
```
Accessing members from a Pointer

• Can dereference the pointer first then use the dot operator
• Unfortunately, . has higher precedence than * requiring you to use parenthesis

```cpp
#include<iostream>
using namespace std;
enum {CSCI=1, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[])
{
    student s1, *stu_ptr;
    strncpy(s1.name,"Jill",80);
    s1.id = 5; s1.major = CECS;
    stu_ptr = &s1;
    (*stu_ptr).id = 4;
    strncpy( (*stu_ptr).name, "Tom",80);

    return 0;
}
```
When Are Pointers To Objects Used?

- Pointers to object occur commonly when they are passed by reference or dynamically allocated.

```cpp
#include<iostream>
using namespace std;
enum {CSCI=1, CECS };

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

void modifyStudent(student* stuptr);

int main(int argc, char *argv[])
{
    student* stu = new student;
    strncpy((*stu).name,"Jill",80);
    (*stu).id = 5; (*stu).major = CECS;

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

- Save keystrokes & have cleaner looking code by using the arrow (->) operator
  - (*struct_ptr).member equivalent to struct_ptr->member
  - Always of the form: ptr_to_struct->member

```cpp
#include<iostream>
using namespace std;
enum {CSCI=1, CECS};

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

int main(int argc, char *argv[]) {
    student *stu = new student;
    strncpy(stu->name, "Jill", 80);
    stu->id = 5; stu->major = CECS;
    ...
    delete stu;
    return 0;
}
```
Passing Objects 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]
- Objects 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 objects since pass by value means making a copy of a large objects]

```c
#include<iostream>

using namespace std;

struct Point {
    int x;
    int y;
};

void print_point(Point myp) {
    cout << "(x,y)=" << myp.x << "," << myp.y;
    cout << endl;
}

int main(int argc, char *argv[]) {
    Point p1;
    p1.x = 2; p1.y = 5;
    print_point(p1);
    return 0;
}
```
Returning Objects

- Can return a struct from a function
- Will return a *copy* of the struct indicated – i.e. 'return-by-value'

```cpp
#include<iostream>

using namespace std;

struct Point {
    int x;
    int y;
};

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

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

int main(int argc, char *argv[]) {
    Point p1;
    p1 = make_point();
    print_point(&p1);
    return 0;
}
```
C++ STRINGS
C Strings

• In C, strings are:
  – Character arrays (char mystring[80])
  – Terminated with a NULL character
  – Passed by reference/pointer (char *) to functions
  – Require care when making copies
    • Shallow (only copying the pointer) vs. Deep (copying the entire array of characters)
  – Processed using C String library (<cstring>)
String Function/Library (cstring)

- int strlen(char *dest)
- int strcmp(char *str1, char *str2);
  - Return 0 if equal, >0 if first non-equal char in str1 is alphanumerically larger, <0 otherwise
- char *strcpy(char *dest, char *src);
- strncpy(char *dest, char *src, int n);
  - Maximum of n characters copied
- char *strcat(char *dest, char *src);
- strncat(char *dest, char *src, int n);
  - Maximum of n characters concatenated plus a NULL
- char *strchr(char *str, char c);
  - Finds first occurrence of character ‘c’ in str returning a pointer to that character or NULL if the character is not found

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

int main() {
    char temp_buf[5];
    char str[] = "Too much";
    strcpy(temp_buf, str); // bad
    strncpy(temp_buf, str, 4);
    temp_buf[4] = '\0';
    return 0; }
```

In C, we have to pass the C-String as an argument for the function to operate on it.
Recall our conversation of shallow vs. deep copies

Can we just use the assignment operator, ‘=’ with character arrays?
  - No, must allocate new storage

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

char *names[10];

int main()
{
    char temp_buf[100];

    cin >> temp_buf;
    // user enters "Timothy"
    names[0] = temp_buf;

    cin >> temp_buf;
    // user enters "Christopher"
    names[1] = temp_buf;

    return 0;
}
```
Copying Strings/Character Arrays in C

- Must allocate new storage

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

char *names[10];

int main()
{
    char temp_buf[100];
    cin >> temp_buf;
    // user enters "Timothy"
    names[0] = new char[strlen(temp_buf)+1];
    strcpy(names[0], temp_buf);

    cin >> temp_buf;
    // user enters "Christopher"
    names[1] = new char[strlen(temp_buf)+1];
    strcpy(names[1], temp_buf);

    return 0;
}
```
C++ Strings

• So you don't like remembering all these details?
  – You can do it! Don't give up.

• C++ provides a 'string' class that **abstracts** all those worrisome details and **encapsulates** all the code to actually handle:
  – Memory allocation and sizing
  – Deep copy
  – etc.
C++ Strings

• In C++, the `string` class provides an easier alternative to working with plain-old character arrays

• Do's and Don'ts
  – **Do** `#include <string>` and put using namespace std;
  – **Do** initialize using = or by giving an initial value in parentheses (aka use the "constructor" syntax)
  – **Don't** need to declare the size (i.e. `[7]`), just assign
  – **Do** still use it like an array by using `[index]` to get individual characters
  – **Do** still use `cin`/`cout` with strings
  – **Don't** worry about how many characters the user types when inputting to a C++ string

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

int main()
{
    char str1[7] = "CS 103";
    // * Initializes the array to "CS 103" *
    string str2 = "CS 103";
    string str3("Hello");  // constructor
    // * Initializes str2 to "CS 103" &
    //    str3 to "Hello" *

    str2[5] = '4';  // now str2 = "CS 104"
    cout << str2 << endl;
    // prints "CS 104"

    cin >> str1;  // If the user types more
    // than 6 chars..uh oh!
    cin >> str2;  // str2 will adjust to
    // hold whatever the user
    // types
    return 0;
}
```
What Happens Behind the Scenes

- Strings simply abstract character arrays
- Behind the scenes strings are just creating and manipulating character arrays but giving you a simplified set of operators and functions
- Can concatenate (append) to a string with the + operator

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

int main()
{
    string str2 = "CS 103";
    // str2 stores 6 chars. = "CS 103"

    str2 = "Computer Science";
    // now str2 stores 16 characters

    // Can append using '+' or '+=' operator
    str2 = str2 + " is cool";
    // now str2 stores 24 characters
}
```
String Comparison

- C++ strings will perform lexicographic (alphabetical) comparison when comparison operators (<, >, ==, etc.) are applied
- Comparison operators do not work with plain old character arrays

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

int main()
{
    char str1[4] = "abc";
    string str2 = "abc";
    if( str1 == "abc" ) // doesn't work
        {...}
    if( str2 == "abc" ) // works..true
        {...}
    if( str1 < "aac" ) // doesn't work
        {...}
    if( str2 < "aac" ) // works..false
        {...}
    string str3 = "acb";
    if( str3 > str2 ) // works..true
        {...}
}
```
Calling Member Functions (Methods)

• Use the dot operator to call an operation (function) on an object or access a data value
  
  • Asking for the string size
    – Call the `.size()` function on a string to get the number of characters stored in the string
  
  • Can generate substrings
    – Call either of the 2 versions: `.substr(start_index)` or `.substr(start_index, length)` function on the string

• Many more member functions

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

int main()
{
    string mystr = "CS 103";
    cout << mystr.size() << endl; // 6

    string s = mystr.substr(2);
    // s = "103"

    mystr = "Computer Science";
    cout << mystr.size() << endl; // 16

    s = mystr.substr(9,2);
    // s = "Sc"
}
```

http://www.cplusplus.com/reference/string/string/
Other Member Functions

- Get C String (char *) equiv.
- Find a substring
  - Searches for occurrence of a substring
  - Returns either the index where the substring starts or string::npos
  - std::npos is a constant meaning ‘just beyond the end of the string’...it’s a way of saying ‘Not found’
- Others: replace, rfind, etc.

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

int main() {
    string s1("abcdef");
    char my_c_str[80];
    strcpy(my_c_str, s1.c_str());
    cout << my_c_str << endl;

    size_t idx = s1.find("bcd");
    if(idx != string::npos){
        cout << "Found bcd starting at pos=":
        cout << idx << endl;
    } else {
        cout << "Not found" << endl;
    }
    return 0;
}
```

Output:

```
abcdef
Found bcd starting at pos=1
```
Other Objects We've Used

• You've already used objects
  – ifstream
  – iostream

• Can initialize at declaration by passing initial value in ( )
  – Known as a constructor

• Use the **dot operator** to call an operation (function) on an object or access a data value

• Some special **operators** can be used on certain object types (+, -, [], etc.) but you have to look them up

```cpp
#include <iostream>
#include <fstream>
#include <fstream>
Using namespace std;

int main()
{
    int x; char line[80];
    ifstream myfile(argv[1]);
    if( ! myfile.fail() ){
        myfile >> x;
    }

    cin.getline(line, 80);
    cout << line << endl;
    return 0;
}
```

**cin, cout, ifstreams, and ofstreams are examples of Objects**
Exercises (If Time Allows)

- [http://bits.usc.edu/cs103/in-class-exercises/](http://bits.usc.edu/cs103/in-class-exercises/)
  - Palindrome
  - Circular Shift
ENUMERATIONS
Enumerations

- Associates an integer (number) with a symbolic name
  
  enum [optional_collection_name] {Item1, Item2, ... ItemN}
  - Item1 = 0
  - Item2 = 1
  - ...
  - ItemN = N-1

- Use symbolic item names in your code and compiler will replace the symbolic names with corresponding integer values

```cpp
const int BLACK=0;
const int BROWN=1;
const int RED=2;
const int WHITE=7;

int pixela = RED;   // pixela = 2;
int pixelb = BROWN; // pixelb = 1;
```

Handling symbolic names with given codes

```cpp
// First enum item is associated with 0
enum Colors {BLACK,BROWN,RED,...,WHITE};

int pixela = RED;   // pixela = 2;
int pixelb = BROWN; // pixelb = 1;
```

Using enumeration to simplify
Aliases for a type name

TYPEDEF'S
typedef’s

• Often we do not want to always type so much to declare an instance of a struct
• typedefs allow us to create an ‘alias’ for a data-type.
• Format:
  ```
  typedef official_type alias_name
  ```
• Examples:
  ```
  typedef int score_t;
  typedef double decimal_t;
  // x is really an int, y is really a double
  score_t x; decimal_t y;
  ```
• Can be used to make the use of variables more obvious
  – Imagine you had a few int's being used as scores and many other ints elsewhere...then you have to change all score variables to doubles. You can't just find..replace all ints to doubles.