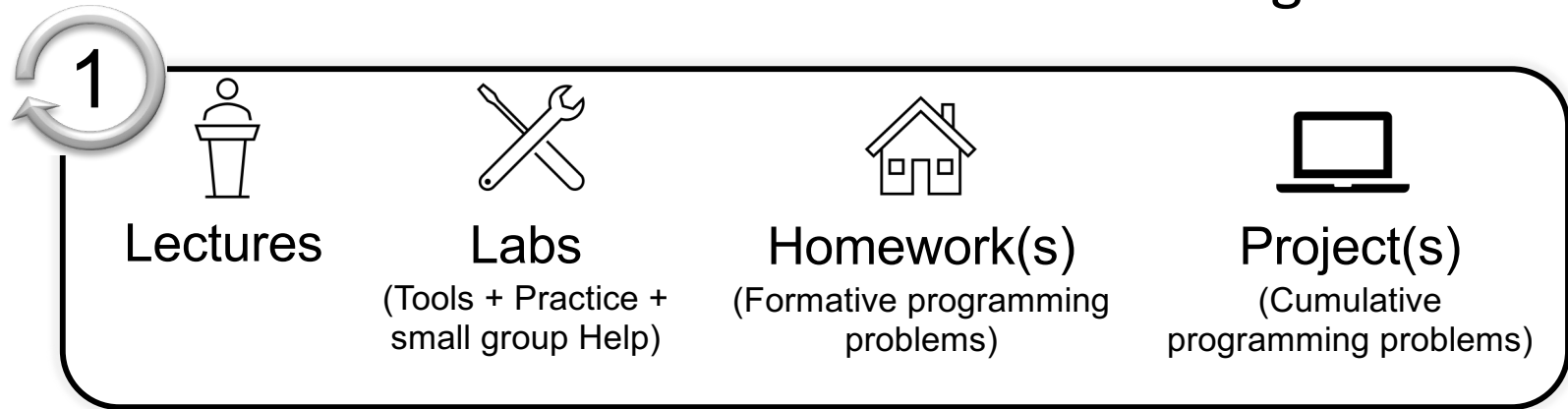


CS 103 Unit 3a – Objects Intro: Structs and Strings

CSCI 103L Teaching Team

Course Structure

- The course is broken into 6 units each consisting of:



**C++ Language
Syntax**



Pointers and Memory



Objects 1



Managing Data



Objects 2



Recursion

Make arbitrary integer codes more readable

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

```
const int BLACK=0;
const int BROWN=1;
const int RED=2;

const int WHITE=7;

int pixela = RED;
int pixelb = BROWN;
...
```

Hard coding symbolic names with given codes

```
// 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

```
enum {CSCI, CSBA, CECS, CSGM, AMCM, QBIO};

int major = AMCM;    // major = 4;
int minor = CSCI;    // minor = 0;
```



Use enumerations to make related integer codes/constants more readable.

OBJECTS

Review: Program Decomposition

- C is a **procedural** language
 - A **function** or **procedure** is the primary unit of code organization, problem decomposition, and abstraction
 - **Functions** can be reused across many applications
- C++ is considered an **object-oriented** language (adds objected-oriented constructs to C) though still supports a procedural approach
 - A **class** or **object** is the primary unit of code organization, problem decomposition, and abstraction
 - Can be reused



This Photo by Unknown Author is licensed under [CC BY-NC](#)

Exercise

- To decompose a program into **functions**, try listing the **verbs** or **tasks** that are performed to solve the problem
 - Model a card game as a series of tasks/procedures...
 - A database representing a social network
- To decompose a program into **objects**, listen for the **nouns**, **objects**, or agents that are interacting

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 entity
- 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

Objects

- Often times we want to represent higher level concepts, objects, or things (beyond an integer, character, or double)
 - Examples: a pixel, a circle, a student, a file
- 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 (embodied 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)

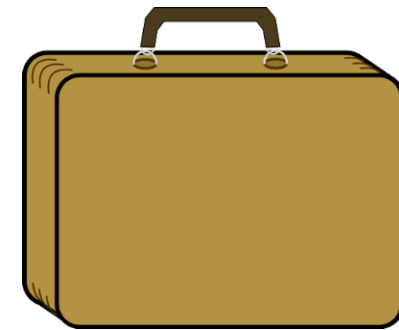
Motivation for Objects

- When going to the airport, would you rather carry all your luggage items piecemeal or pack everything in one suitcase?



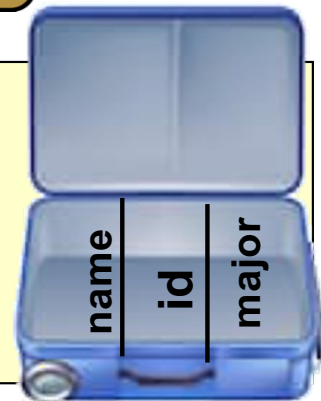
```
void process(char name[], int id, int major)
{
}

```



```
void process(Student s1)
{
}

```



s1

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()`

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++ and now behave almost EXACTLY the same as a **class** (i.e. struct/class are nearly interchangeable)
- **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'
 - We'll look at **structs** now, and **classes** a bit later, but recall they are interchangeable. So pay attention now to make subsequent lectures easier.

Starting with data...

STRUCTS

Types and Instances

- A '**type**' indicates **how much memory** will be required, **what the bits mean** (i.e. integer, double, pointer), and **what operations** can be performed
 - **int** = 32-bits representing only integer values and supporting +, -, *, /, =, ==, <, >, etc.
 - **char*** = 64-bits 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;`

Definitions and Instances (Declarations)

- Objects must first be defined/declared (as a 'struct' or 'class')
 - The declaration is a blueprint that indicates what any instance should look like
 - Identifies the overall name of the struct and its individual member types and names
 - The declaration does not actually create a variable (no memory is allocated)
 - Usually appears outside any function
- Once declared, any number of instances can be created/instantiated in your code
 - **Instances** are actual objects (memory is allocated for each) created from the definition (blueprint)
 - Declared like other variables

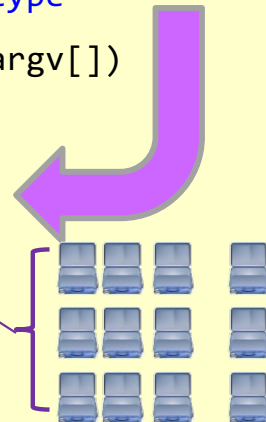
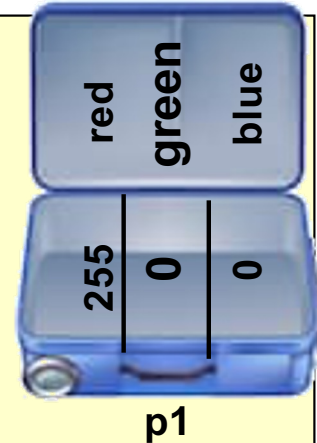
```
#include <iostream>

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 p1;
    pixel image[256][256];
    // make p1 red
    p1.red = 255;
    p1.blue = p1.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 earlier in the object 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

```
#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;

    // how would you set their name to
    // "Tina"

    my_student.id = 1682942;
    my_student.major = CSCI;
    ...
    return 0;
}
```

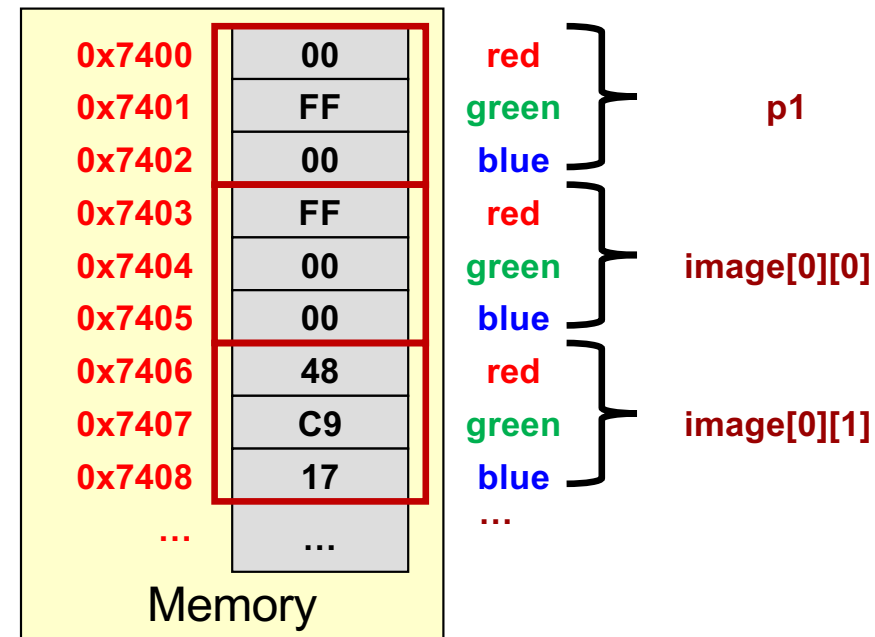


Memory View of Objects

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

```
#include<iostream>

using namespace std;
// declaration (blueprint)
struct pixel {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};
int main(int argc, char *argv[])
{
    int i,j;
    // instantiations (object allocation)
    pixel p1;
    pixel image[256][256];
    ...
    return 0;
}
```



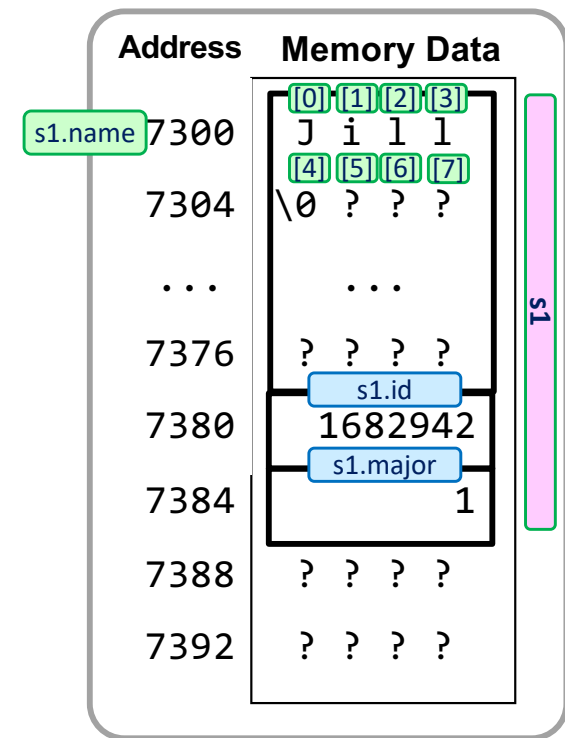
Memory View of Objects

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

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

// declaration (blueprint)
struct student {
    char name[80];
    int id;
    int major;
};

int main(int argc, char *argv[])
{
    int i,j;
    // instantiation (object allocation)
    student s1;
    ...
    return 0;
}
```



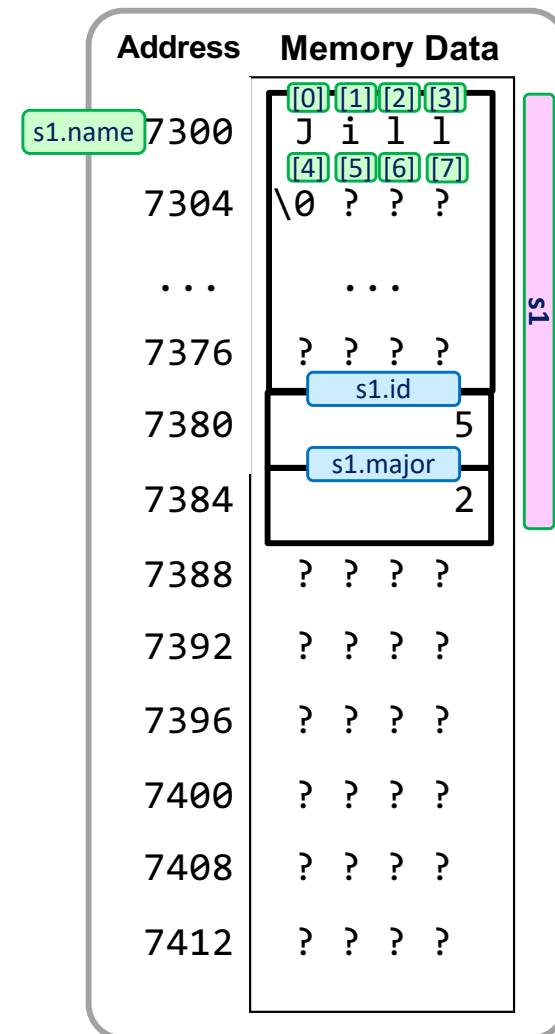
Assignment semantics and pointers to objects

IMPORTANT NOTES ABOUT OBJECTS

Object assignment

- Consider the following initialization of s1

```
#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 a **member-by-member copy** of the entire source object to the destination object

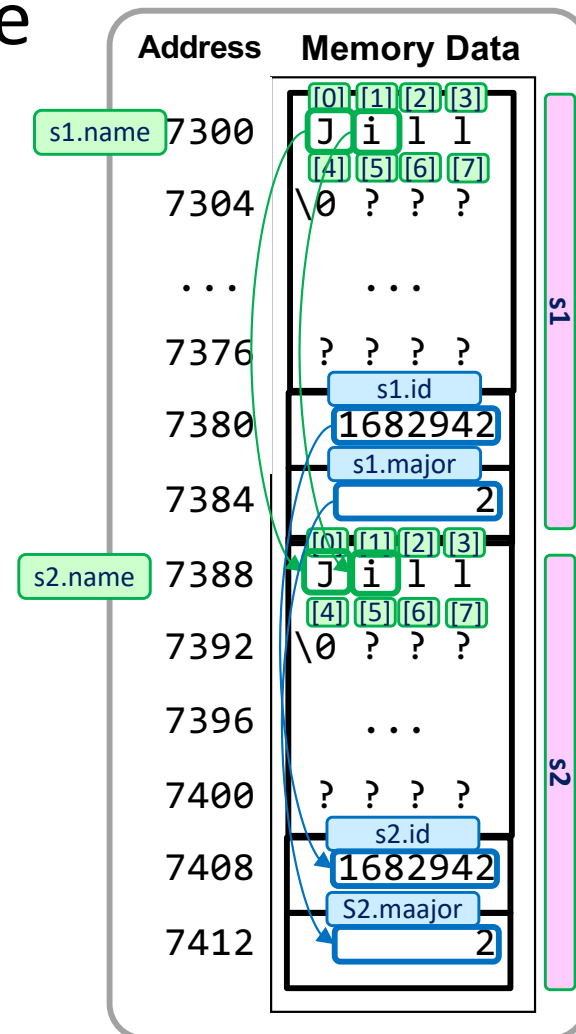
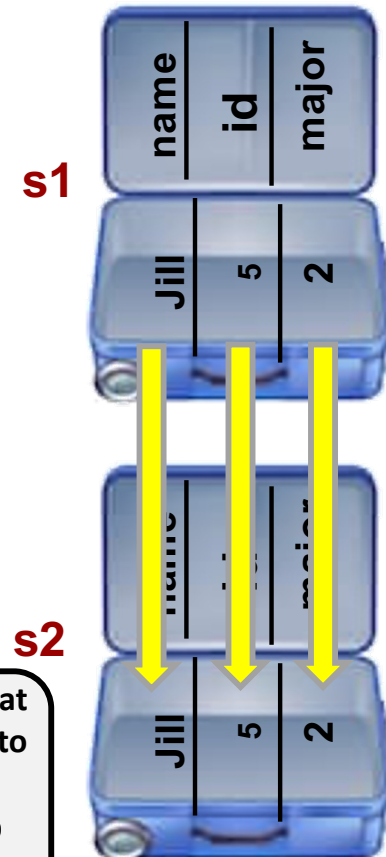
```
#include<iostream>
using namespace std;

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;
}
```

Normally, C/C++ perform only one operation at a time, and make you write code if you want to do more. But object assignment is an exception (probably because we can't "loop through" the different members of a struct).

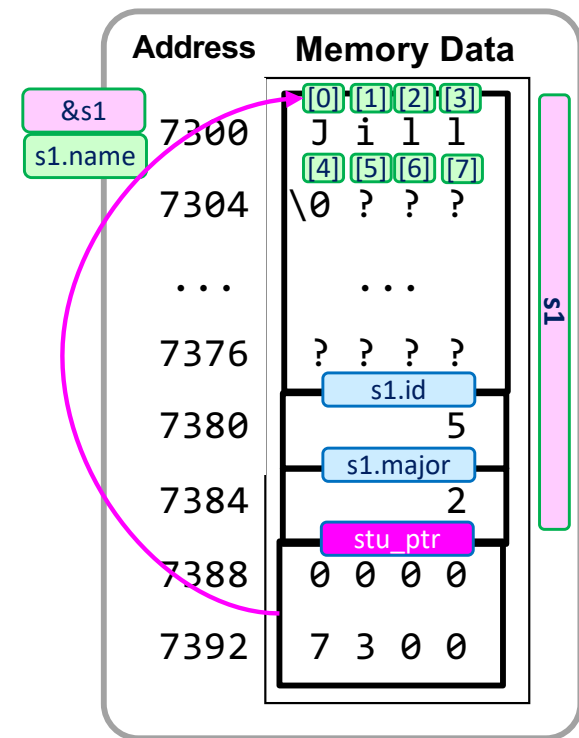


Pointers to Objects

- We can declare **pointers** to objects just as any other variable
- The address of a struct is just (you guessed it) its starting address

```
#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;
    student *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

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

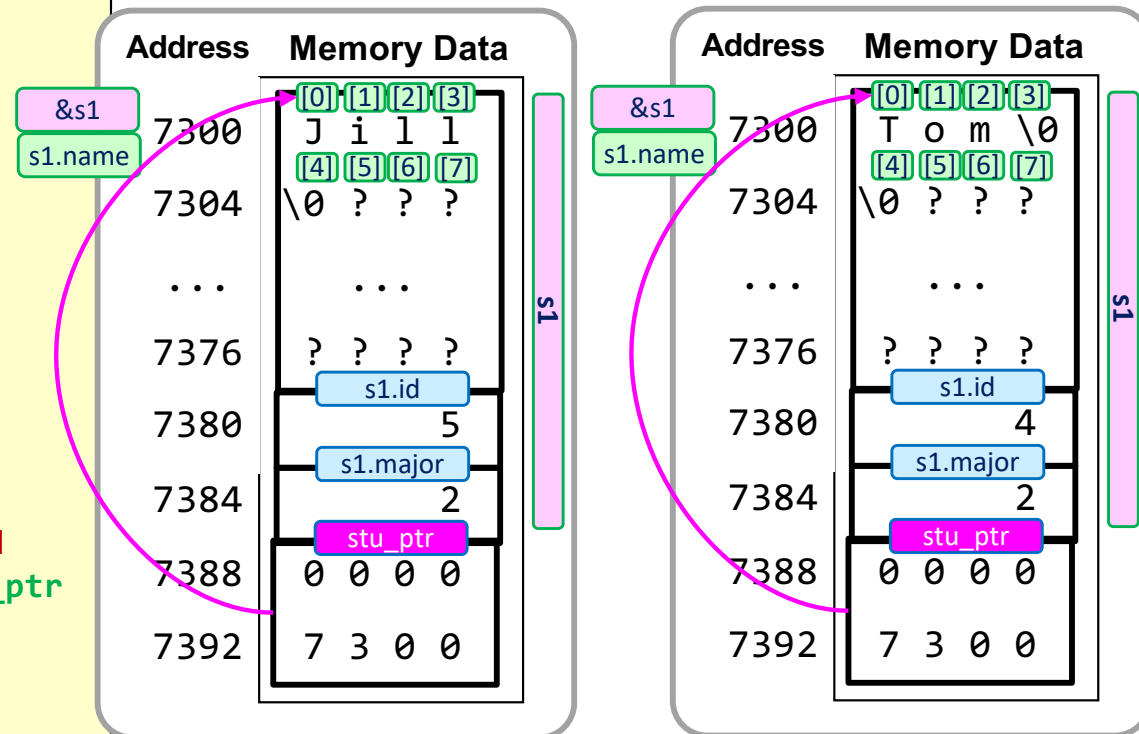
enum {CSCI=1, CECS };

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;    // incorrect - derefs id
    (*stu_ptr).id = 4;  // correct - derefs stu_ptr
    strncpy( (*stu_ptr).name, "Tom",80);

    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

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

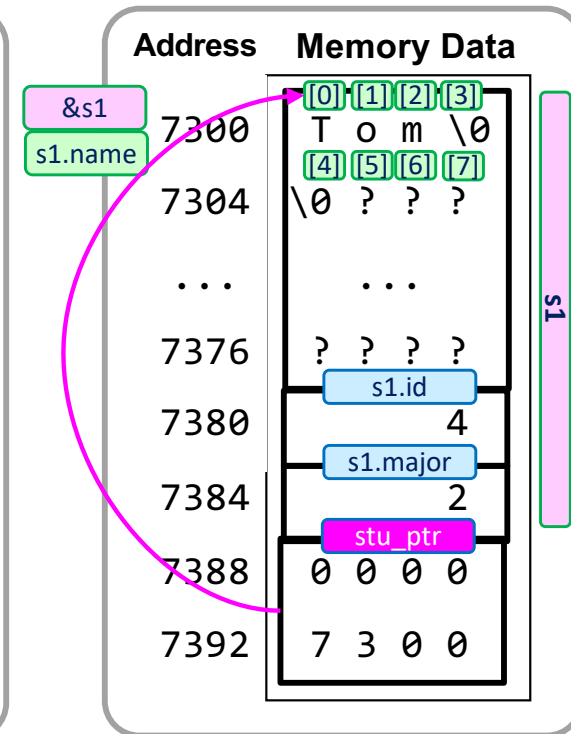
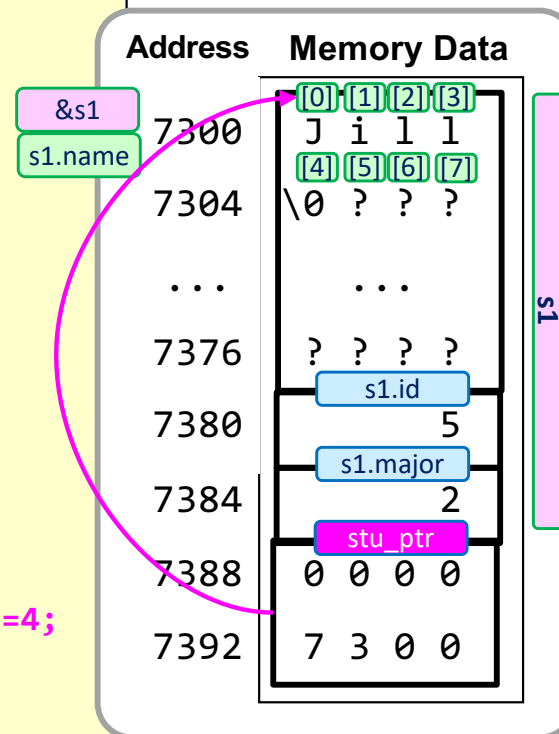
enum {CSCI=1, CECS };

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;    // same as (*stu_ptr).id=4;
    strncpy( stu_ptr->name, "Tom",80);

    return 0;
}
```



When Are Pointers To Objects Used?

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

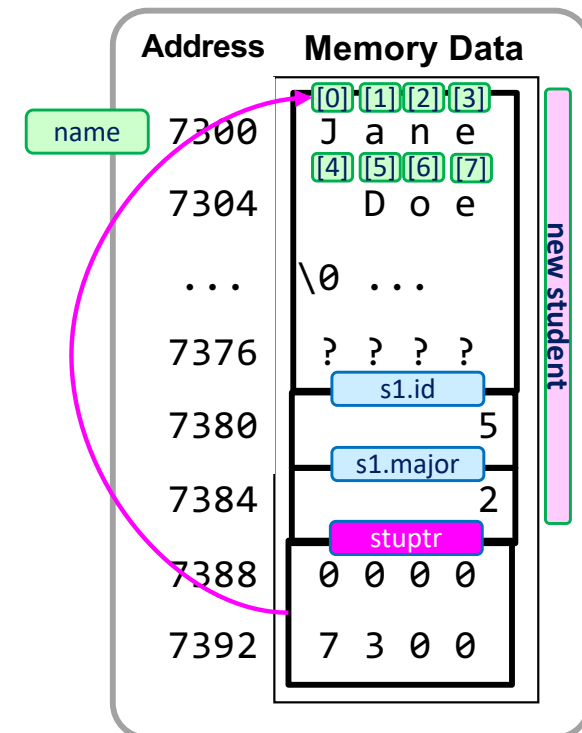
struct student {
    char name[80];
    int id;
    int major;
};

student* makeStudent(const char* n, int i, int m)
{
    student* stuptr = new student;

    strncpy(stuptr->name, n, 80);
    stuptr->id = i; stuptr->major = m;
    return stuptr;
}

int main()
{
    student* stuptr =
        makeStudent("Jane Doe", 5, CECS);
    ...
    cout << stuptr->name << endl;
        // prints "Jane Doe"
    delete stuptr;
    return 0;
}
```

- Pointers to objects occur commonly when objects are **passed by reference** or **dynamically allocated**



Passing Objects as Arguments

- In C, arguments must be a single value [i.e. 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 member-by-member 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]

```
#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'

```
#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

Motivation

- Before we dive deeper into writing our OWN objects...
- ...Let's learn how to use objects (structs/classes) that the C++ library already provides us.
- One of the most basic (and useful) objects in the C++ library is the `string` class.
- To understand what they do for you (and how they provide "encapsulation" and "abstraction", let's review how we dealt with strings in C (i.e. before the `string` class existed)

Review: C Strings

- In C, strings are:
 - Character arrays (`char mystring[80];`)
 - Terminated with a NULL character (`'\0' ⇔ 0`)
 - 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

In C, we have to pass the C-String as an argument for the function to operate on it

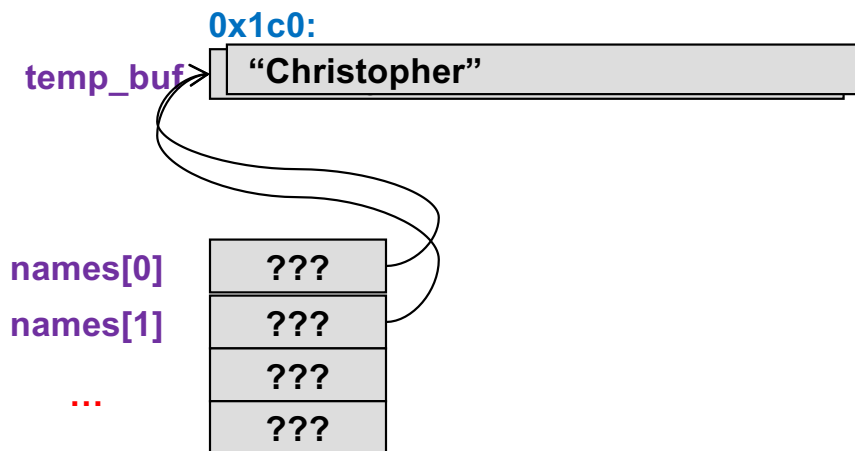
```
#include <cstring>
using namespace std;

int main() {
    char temp_buf[5];
    char str[] = "Too much";

    strcpy(temp_buf, str); // bad
    return 0;
}
```

Review: Shallow vs. Deep C-String Copy

- Recall our conversation of shallow vs. deep copies
- Can we just use the assignment operator, '=', with character arrays?
 - No, must allocate new storage



Assigning an array name just assigns a pointer and does NOT make a copy of the array.

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

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];

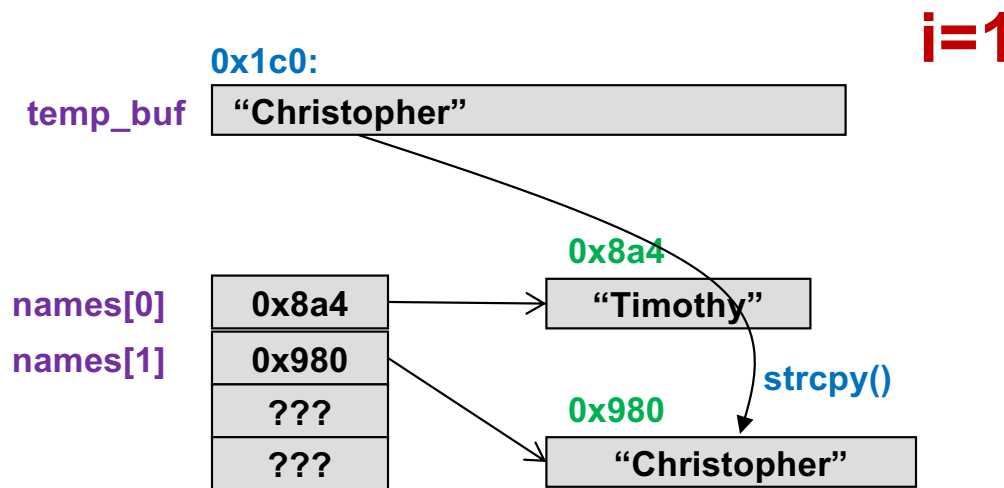
    // One "scratchpad" array for all inputs
    char temp_buf[40];

    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = temp_buf;
    }
    // Do stuff with names

    for(int i=0; i < 10; i++){
        delete [] names[i];
    }
    return 0;
}
```

More Dealing with Text Strings

- Must allocate new storage



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

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];

    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        // Find length of strings
        int len = strlen(temp_buf);
        names[i] = new char[len + 1];
        strcpy(names[i], temp_buf);
    }

    // Do stuff with names

    for(int i=0; i < 10; i++){
        delete [] names[i];
    }

    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
 - Concatenation, Comparison, Size information
 - 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

```
#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
}
```

What Happens Behind the Scenes

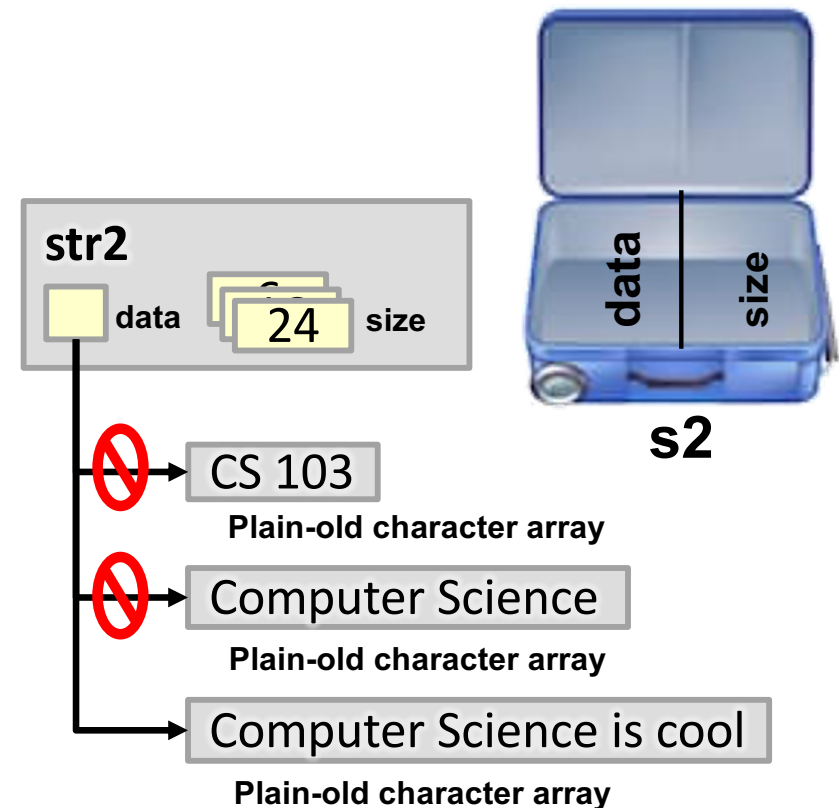
- Strings simply abstract character arrays
- Behind the scenes strings are just creating and manipulating character arrays but giving you a simplified set of operators and functions
- Can concatenate (append) to a string with the + operator

```
#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

```
#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

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

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

    string s = mystr.substr(3);
    // s = "103"

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

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

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.

```
#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
```

String ↔ Number Conversion



- Recall: Casting does **NOT** work for **string** ↔ **numeric** conversions.
- Instead, use functions defined in `<string>`
- Conversion from **number** to **string**:
 - `string to_string(int);`
 - `string to_string(double);`
 - ...
- Conversion from **string** to **number**:
 - `int stoi(string);`
 - `unsigned int stoul(string);`
 - `double stod(string);`



```
#include <iostream>
#include <string>
using namespace std;
int main() {

    double a = 3.6;
    int b = static_cast<int>(a) / 2;
        // Works! b = 1 (casts 3.6 to 3)

    int c = 123;
    string d = static_cast<string>(c);
        // Error! Doesn't compile.
    string d = to_string(c);
        // Works! But only since C++11

    string e = "42";
    int f = static_cast<int>(e);
        // Error! Doesn't compile.
    int f = stoi(e); // string-to-int
        // Works! But only since C++11
        // use stod() for string-to-double
    return 0;
}
```

Summary

- You've already used objects
 - iostream (cin and cout)
 - Now file streams and strings
 - There will be many more objects we can use from the C++ library
- 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

```
#include <iostream>
#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)

- Palindrome
- Circular Shift