CS 103 Unit 9 – Objects, Structs, and Strings

Mark Redekopp

OBJECTS

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;

Abstract Data Types

- Often times we want to represent abstract things (beyond an integer, character, or double)
  - Examples:
    - A pixel, a circle, a student
- Often these abstract types 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 larger 'thing' as well as supporting operations on that 'thing'
  - Can reference the collection with a single name (pixelA, student1)
  - Can access individual components (pixelA.red, student1.id)

Objects

- An object is a group of data + functions
- 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()

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.

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
    - Place data and code that operates on that data together into one unit
    - 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

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)

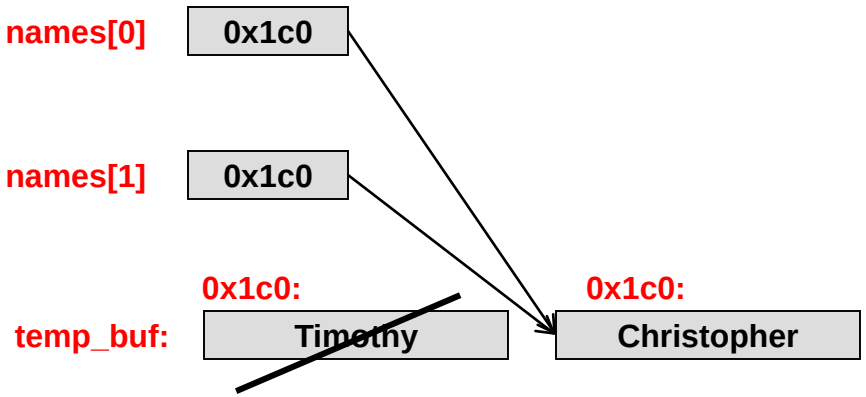
In C, we have to pass the C-String as an argument for the function to operate on it

- 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

```
#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; }
```

Copying Strings/Character Arrays in C

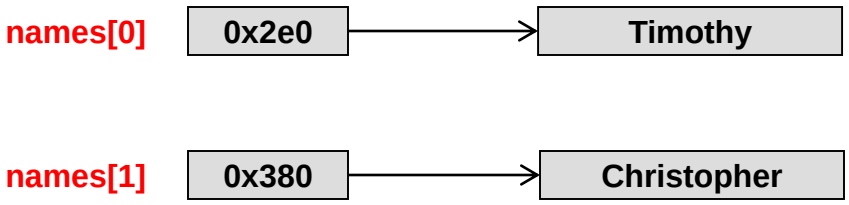
- Recall our conversation of shallow vs. deep copies
- Can we just use the assignment operator, '=' with character arrays?
- No, must allocate new storage




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

// store 10 user names of up to 80 chars
//   names type is still char **
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

- No, must allocate new storage




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

// store 10 user names of up to 80 chars
//   names type is still char **
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.

Object Syntax Overview

- You've already used objects
  - ifstream
  - string
- 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 <string>
using namespace std;

int main(int argc, char *argv[]) {

  // similar to char s1[] = "CS 103"
  string s1("CS 103");

  // len will have 6
  int len = s1.size();

  // s2 will have "103"
  string s2 = s1.substr(3,3);

  // s3 will have "CS 103 is fun"
  string s3 = s1 + " is fun";

  // will print 'C'
  cout << s1[0] << endl;
  return 0;
}
```

String and Ifstreams are Examples of Objects

```
ifstream myfile(argv[1]);

myfile.fail();

myfile >> x;
```

String Examples

- 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)

http://www.cplusplus.com/reference/string/string/

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

int main(int argc, char *argv[]) {
  int len;
  string s1("CS 103");
  string s2 = "fun";

  s2 = "really fun";

  cout << s1 << " is " << s2 << 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:

CS 103 is really fun
really fun!!!
CS 103 same as CS 103
First letter is C

More String Examples

- Size/Length of string
- 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'
- Get a substring
  - Pass it the start character index and the number of characters to copy
  - Returns a new string
- Others: replace, rfind, etc.

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

int main(int argc, char *argv[]) {
  string s1("abc def");
  cout << "Len of s1: " << s1.size() << endl;

  char my_c_str[80];
  strcpy(my_c_str, s1.c_str() );
  cout << my_c_str << 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
abc def
The string is: abc def
Found bc_d starting at pos=1
def
```

http://www.cplusplus.com/reference/string/string/

Exercises

- http://bits.usc.edu/cs103/in-class-exercises/
  - Palindrome
  - Circular Shift

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

```
#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**

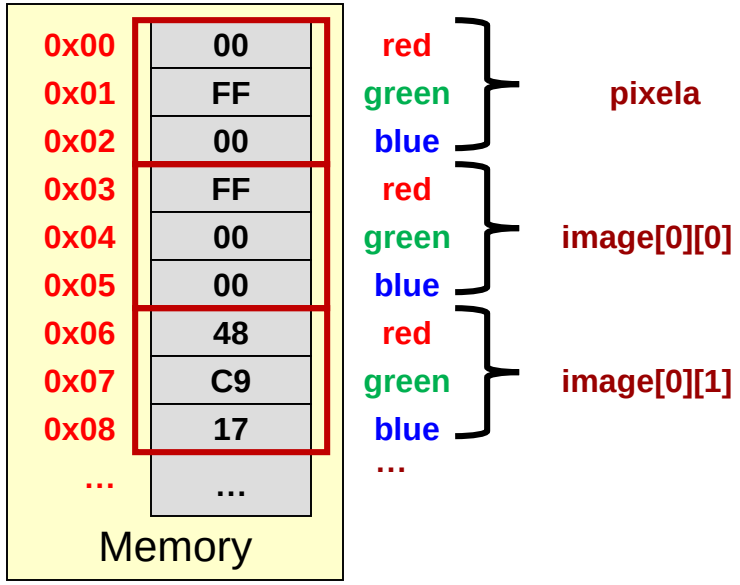
```
#include<iostream>
using namespace std;

enum {CS, CECS};

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 = CS;
  if(my_student.major == CECS)
    cout << "You like HW" << endl;
  else
    cout << "You like SW" << endl;
  ...
  return 0;
}
```

*80 + 8 + 8 bytes in struct, but only allocate when it is called in main

Memory View of Objects

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

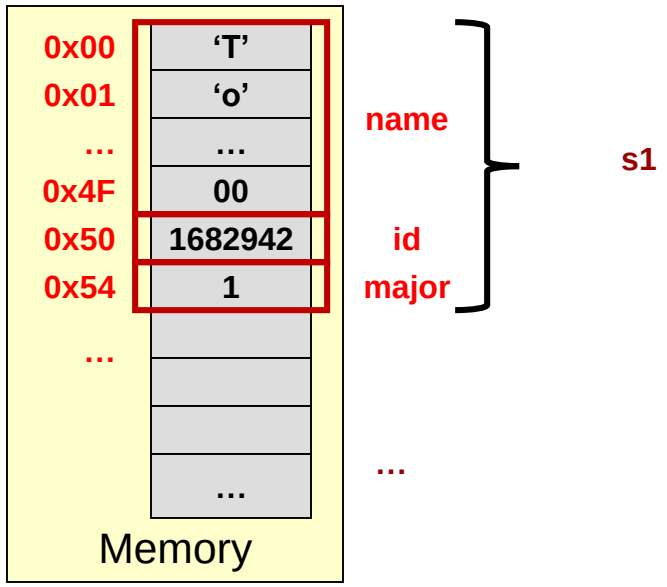
```
#include<iostream>

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

```
#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;
}
```



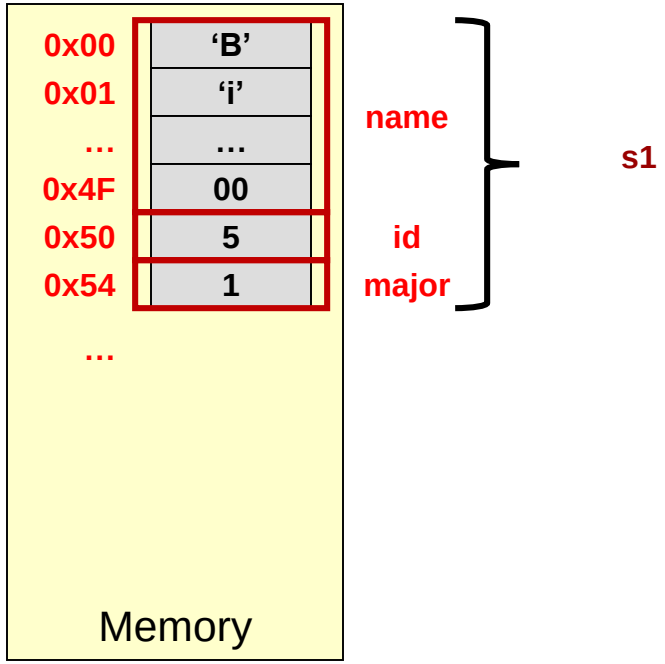
Assignment semantics and pointers to objects

IMPORTANT NOTES ABOUT OBJECTS

Object assignment

- Consider the following initialization of s1

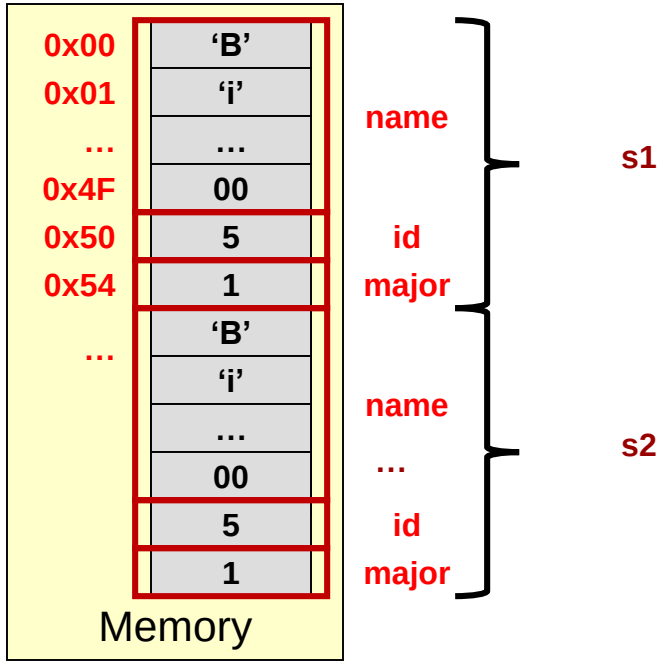
```
#include<iostream>
using namespace std;

enum {CS, CECS};

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

int main(int argc, char *argv[])
{
  student s1,s2;
  strncpy(s1.name,"Bill",80);
  s1.id = 5; s1.major = CECS;
```



Object assignment

- Assigning one object to another will perform an element by element copy of the source struct to the destination object

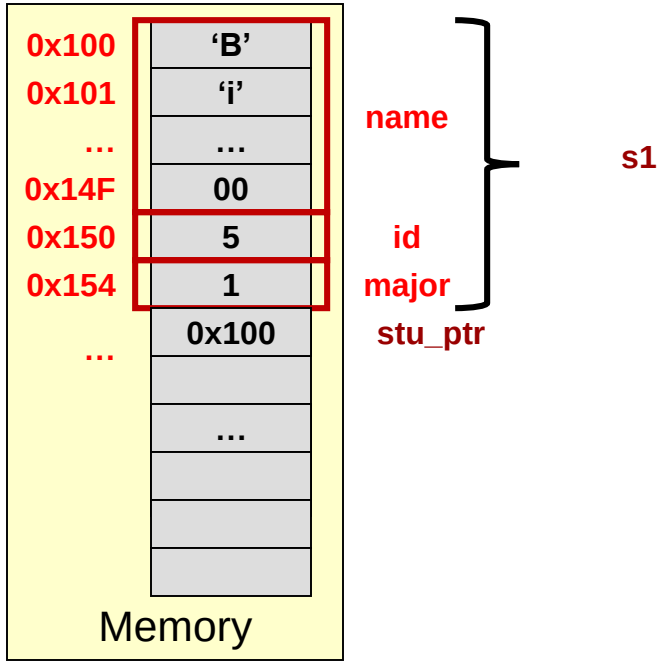
```
#include<iostream>
using namespace std;

enum {CS, CECS };

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

int main(int argc, char *argv[])
{
  student s1,s2;
  strncpy(s1.name,"Bill",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

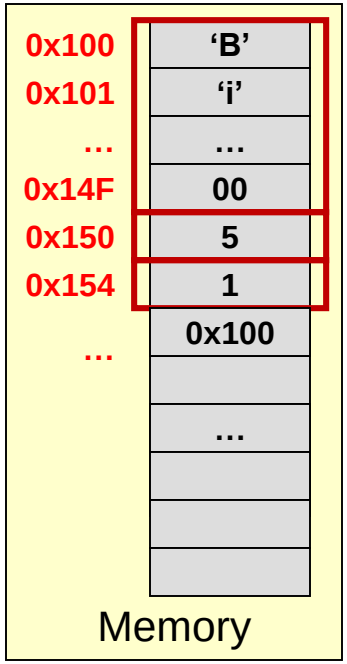
```
#include<iostream>
using namespace std;

enum {CS, CECS };

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

int main(int argc, char *argv[])
{
  student s1, *stu_ptr;
  strncpy(s1.name,"Bill",80);
  s1.id = 5; s1.major = CECS;
  stu_ptr = &s1;
  return 0;
}
```

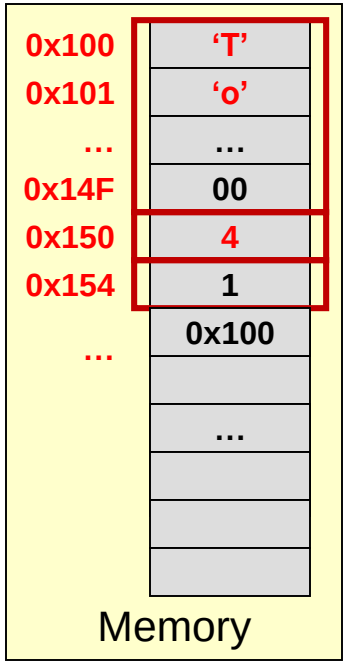


Accessing members from a Pointer

- Can dereference the pointer first then use the dot operator

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

enum {CS, CECS };

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

int main(int argc, char *argv[])
{
  student s1,*stu_ptr;
  strncpy(s1.name,"Bill",80);
  s1.id = 5; s1.major = CECS;
  stu_ptr = &s1;
  (*stu_ptr).id = 4;
  strncpy( (*stu_ptr).name, "Tom",80);


  return 0;
}
```

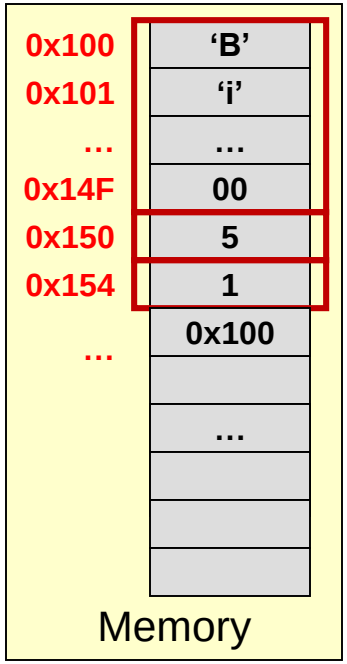


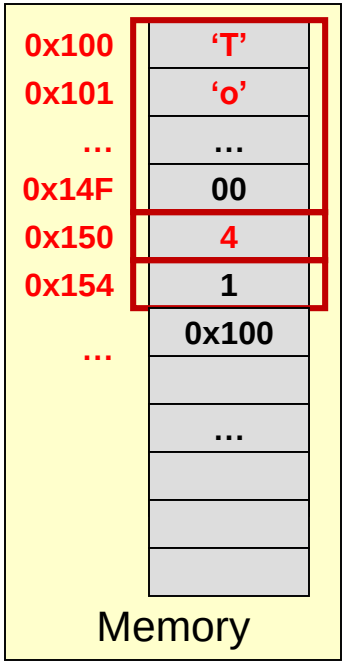


Arrow (->) operator

- 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_name`

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

enum {CS, CECS };

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

int main(int argc, char *argv[])
{
  student s1,*stu_ptr;
  strncpy(s1.name,"Bill",80);
  s1.id = 5; s1.major = CECS;
  stu_ptr = &s1;
  stu_ptr->id = 4;
  strncpy( stu_ptr->name, "Tom",80);
  ...
  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]

```
#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 only return a single struct from a function [i.e. not an array of objects]
- 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;
}
```

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

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.