C Programming Essentials
Unit 3: Basic Data Structures

CHAPTER 10: STRUCT, UNION AND ENUM

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Be able to use **compound data structures** in programs
- Be able to pass compound data structures as function arguments, either by value or by reference
- Be able to do simple **bit-vector** manipulations

LECTURE 1

enum

enum

- In C programming, an enumeration type (also called **enum**) is a data type that consists of integral constants. To define enums, the **enum** keyword is used.

```
enum flag {const1, const2, ..., constN};   // enum example
```

- By default, const1 is 0, const2 is 1 and so on. You can change default values of enum elements during declaration (if necessary).

enum Example

```
// Changing default values of enum constants
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3,
};
```

Enumerated Type Declaration

- When you define an enum type, the blueprint for the variable is created. Here's how you can create variables of enum types.

```
enum boolean {false, true};
enum boolean check; // declaring an enum variable
```

- Here, a variable check of the type enum boolean is created.
- You can also declare enum variables like this.

```
enum boolean {false, true} check;
```

Demo Program: enum1.c

Day 4

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main(){
    // creating today variable of enum week type
    enum week today;
    today = Wednesday;
    printf("Day %d",today+1);
    return 0;
}
```

Why enums are used?

An enum variable can take only one value. Here is an example to demonstrate it (Demo Program: enum2.c)

```
#include <stdio.h>
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3
} card;

int main() {
    card = club;
    printf("Size of enum variable = %d bytes", sizeof(card));
    return 0;
}
```

```
Size of enum variable = 4 bytes
```

How to use enums for flags?

- Let us take an example,

```
enum designFlags {
  ITALICS = 1,
  BOLD = 2,
  UNDERLINE = 4
} button;
```

- Suppose you are designing a button for Windows application. You can set flags ITALICS, BOLD and UNDERLINE to work with text.

How to use enums for flags?

- There is a reason why all the integral constants are a power of 2 in the above pseudocode.

```
// In binary
ITALICS   = 00000001
BOLD      = 00000010
UNDERLINE = 00000100
```

How to use enums for flags?

- There is a reason why all the integral constants are a power of 2 in the above pseudocode.

```
// In binary
ITALICS   = 00000001
BOLD      = 00000010
UNDERLINE = 00000100
```

How to use enums for flags?

- Since the integral constants are a power of 2, you can combine two or more flags at once without overlapping using bitwise OR | operator. This allows you to choose two or more flags at once. For example,

Demo Program: enum3.c

```c
#include <stdio.h>
enum designFlags {
   BOLD = 1,
   ITALICS = 2,
   UNDERLINE = 4
};
int main() {
   int myDesign = BOLD | UNDERLINE;
      //    00000001
      //  | 00000100
      //  ___________
      //    00000101
   printf("%d", myDesign);
   return 0;
}
```

5

When the output is 5, you always know that bold and underline is used.

How to use enums for flags?

- Also, you can add flags according to your requirements.

```
if (myDesign & ITALICS) {
    // code for italics
}
```

- Here, we have added italics to our design. Note, only code for italics is written inside the if statement.
- You can accomplish almost anything in C programming without using enumerations. However, they can be pretty handy in certain situations.

Basic Types and Operators

Basic data types:

- Types: *char, int, float and double*
- Qualifiers: *short, long, unsigned, signed, const*

Constant: 0x1234, 12, “Some string”

Enumeration:

- Names in different enumerations must be distinct
- ```
  enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};
  enum WeekendDay_t {Sat = 0, Sun = 4};
  ```
  ```

Basic Types and Operators

Arithmetic: +, -, *, /, %

- prefix ++i or --i ; increment/decrement before value is used
- postfix i++, i--; increment/decrement after value is used

Relational and logical: <, >, <=, >=, ==, !=, &&, ||

Bitwise: &, |, ^ (xor), <<, >>, ~(ones complement)

LECTURE 2

struct

Structures

Compound data:

A date is

- an `int month` <u>and</u>
- an `int day` <u>and</u>
- an `int year`

```
struct ADate {
   int  month;
   int  day;
   int  year;
};

struct ADate date;

date.month = 1;
date.day = 18;
date.year = 2018;
```

Unlike Java, C doesn’t automatically define functions for initializing and printing …

Structure Representation & Size

`sizeof(struct …) =`

sum of `sizeof(field)`

\+ alignment padding

Processor- and compiler-specific

```
struct CharCharInt {
    char  c1;
    char  c2;
    int   i;
} foo;

foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xDEADBEEF;
```

c1	c2	padding		i			
61	62			EF	BE	AD	DE

x86 uses “little-endian” representation

How to define structures?

Before you can create structure variables, you need to define its data type. To define a struct, the struct keyword is used.

Syntax of struct

```
struct structureName {
    dataType member1;
    dataType member2;
    ...
};
```

Here is an example:

```
struct Person{
    char name[50];
    int citNo;
    float salary;
};
```

Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Create struct variables

- When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.
- Here's how we create structure variables:

Create struct variables

Demo Program: struct1.c

```
#include <stdio.h>
struct Person{
    char name[50];
    int citNo;
    float salary;
};

int main(){
    struct Person person1, person2, p[20];
    return 0;
}
```

Another way of creating a struct variable is:

```
struct Person {
  char name[50];
  int citNo;
  float salary;
} person1, person2, p[20];
```

Access members of a structure

- There are two types of operators used for accessing members of a structure.

  1. **.** Member operator
  2. **->** - Structure pointer operator (dereference)

- Supposed, you want to access the salary of person2. Here's how you can do it.

  person2.salary

```c
#include <stdio.h>
struct Distance{
    int feet;
    float inch;
} dist1, dist2, sum;
```

```c
int main(){
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);
    printf("2nd distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist2.feet);
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);
    // adding feet
    sum.feet = dist1.feet + dist2.feet;
    // adding inches
    sum.inch = dist1.inch + dist2.inch;
    // changing to feet if inch is greater than 12
    while (sum.inch >= 12){
        ++sum.feet;
        sum.inch = sum.inch - 12;
    }
    printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
    return 0;
}
```

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"
```

LECTURE 3

Type Definition

Typedef

Mechanism for creating new type names

- New names are an alias for some other type
- *May* improve clarity and/or portability of the program

```
typedef long int64_t;
typedef struct ADate {
   int month;
   int day;
   int year;
} Date;

int64_t i = 100000000000;
Date d = { 1, 18, 2018 };
```

Overload existing type names for clarity and portability

Simplify complex type names

Constants

There is no array.length

Allow consistent use of the same constant throughout the program

- Improves clarity of the program
- Reduces likelihood of simple errors
- Easier to update constants in the program

```
int array[10];

for (i=0; i<10; i++) {
   …
}
```

Preprocessor directive

Constant names are capitalized by convention

```
#define SIZE 10

int array[SIZE];

for (i=0; i<SIZE; i++) {
   …
}
```

Define once, use throughout the program

Keyword typedef

- We use the typedef keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables.

- **struct Distance** is a struct data type
- **distance** is also a data type

Keyword typedef

This code

```
struct Distance{
    int feet;
    float inch;
};

int main() {
    struct Distance d1, d2;
}
```

is equivalent to

```
typedef struct Distance{
    int feet;
    float inch;
} distances;

int main() {
    distances d1, d2;
}
```

Keyword typedef

- **struct Distance** is a struct data type
- **distance** is also a data type

Nested Structures

You can create structures within a structure in C programming. For example,

```
struct complex {
  int imag;
  float real;
};

struct number { struct complex comp; int integers; } num1, num2;
```

Supposed, you want to set imag of num2 variable to 11. Here's how you can do it:

```
num2.comp.imag = 11;
```

Why structs in C?

- Supposed, you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables name, citNo and salary to store this information.
- What if you need to store information of more than one person? Now, you need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2, etc.
- A better approach would be to have a collection of all related information under a single name Person structure and use it for every person.

LECTURE 4

Instantiation

C Pointers to struct

Here's how you can create pointers to structs.

```
struct name {
  member1;
  member2;
  . .
};

int main() {
  struct name *ptr, Harry;
}
```

Here, ptr is a pointer to struct.

Access members using Pointer

To access members of a structure using pointers, we use the **->** operator.

```
#include <stdio.h>
struct person {
    int age;
    float weight;
};
```

```
int main() {
    struct person *personPtr, person1;
    personPtr = &person1;
    printf("Enter age: ");
    scanf("%d", &personPtr->age);
    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);
    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);
    return 0;
}
```

Access members using Pointer

To access members of a structure using pointers, we use the **->** operator.

- In this example, the address of person1 is stored in the `personPtr` pointer using `personPtr = &person1;`.
- Now, you can access the members of `person1` using the `personPtr` pointer.
- By the way,
  - `personPtr->age` is equivalent to `(*personPtr).age`
  - `personPtr->weight` is equivalent to `(*personPtr).weight`

Dynamic memory allocation of structs

- Before you proceed this section, we recommend you to check C dynamic memory allocation.
- Sometimes, the number of struct variables you declared may be insufficient. You may need to allocate memory during run-time. Here's how you can achieve this in C programming.

Demo Program: person.c

- Check the demo program

```c
#include <stdio.h>
#include <stdlib.h>
struct person {
	int age;
	float weight;
	char name[30];
};

int main() {
	struct person *ptr; int i, n;
	printf("Enter the number of persons: ");
	scanf("%d", &n); // allocating memory for n numbers of struct person
	ptr = (struct person*) malloc(n * sizeof(struct person));
	for(i = 0; i < n; ++i) {
		printf("Enter first name and age respectively: ");
		// To access members of 1st struct person,
		// ptr->name and ptr->age is used
		// To access members of 2nd struct person,
		// (ptr+1)->name and (ptr+1)->age is used
		scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
	}
	printf("Displaying Information:\n");
	for(i = 0; i < n; ++i)
	printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);
	return 0;
}
```

```
Enter the number of persons:  2
Enter first name and age respectively:  Harry 24
Enter first name and age respectively:  Gary 32
Displaying Information:
Name: Harry	Age: 24
Name: Gary	Age: 32
```

Dynamic memory allocation of structs

- In the above example, n number of struct variables are created where n is entered by the user.
- To allocate the memory for n number of `struct person`, we used,

```
ptr = (struct person*)
        malloc(n * sizeof(struct person));
```

- Then, we used the `ptr` pointer to access elements of `person`.

LECTURE 5

Arrays of Structures

Arrays of Structures

Array declaration

Constant

Array index, then structure field

```
Date birthdays[NFRIENDS];

bool
check_birthday(Date today)
{
  int i;

  for (i = 0; i < NFRIENDS; i++) {
    if ((today.month == birthdays[i].month) &&
        (today.day == birthdays[i].day))
      return (true);

  return (false);
}
```

Pointers to Structures

```
Date
create_date1(int month,
             int day,
             int year)
{
  Date d;

  d.month = month;
  d.day   = day;
  d.year  = year;

  return (d);
}
```

Pass-by-reference

```
void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
  d->month = month;
  d->day   = day;
  d->year  = year;
}
```

Copies date

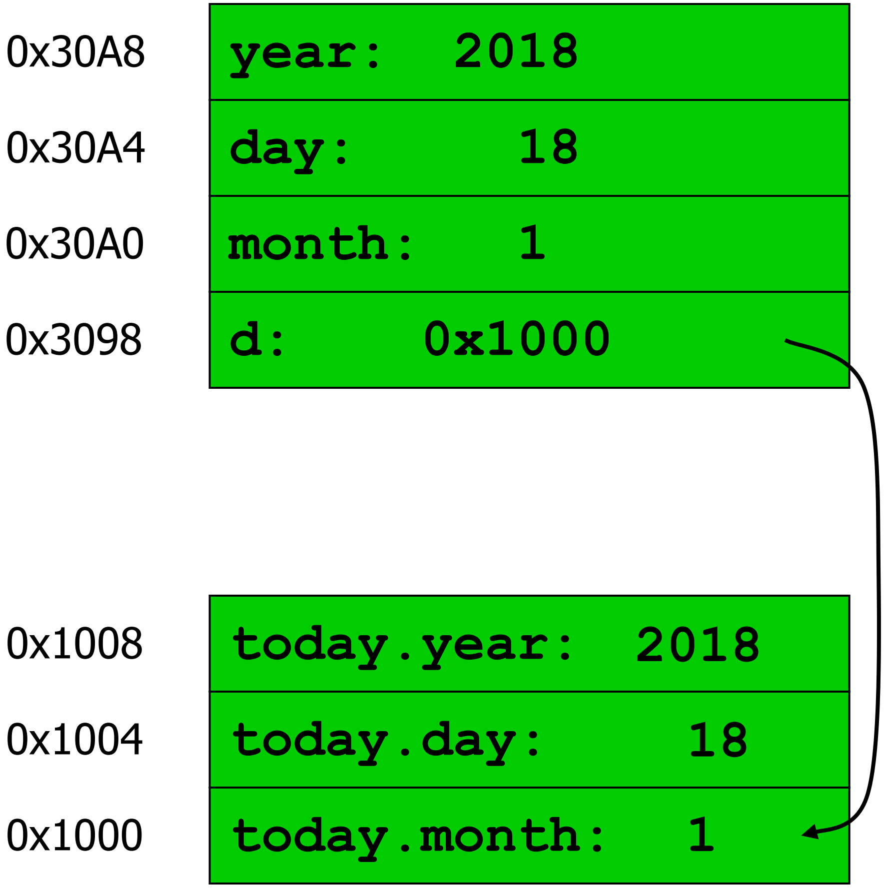
```
Date today;

today = create_date1(1, 18, 2018);
create_date2(&today, 1, 18, 2018);
```

Pointers to Structures

```
void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
  d->month = month;
  d->day   = day;
  d->year  = year;
}

void
fun_with_dates(void)
{
  Date today;
  create_date2(&today, 1, 18, 2018);
}
```



Pointers to Structures

```
Date *
create_date3(int month,
             int day,
             int year)
{
  Date *d;

  d->month = month;
  d->day   = day;
  d->year  = year;

  return (d);
}
```

What is d pointing to?!?!
(more on this later)

Abstraction in C

From the #include file widget.h:

Definition is hidden!

```
struct widget;

struct widget *widget_create(void);
int             widget_op(struct widget *widget, int operand);
void            widget_destroy(struct widget *widget);
```

From the file widget.c:

```
#include "widget.h"

struct widget {
        int x;
        …
};
```

LECTURE 5

C Structure and Function

Passing structs to functions

We recommended you to learn these tutorials before you learn how to pass structs to functions.

- C structures
- C functions
- User-defined Function

Demo Program: student.c

- Here, a struct variable s1 of type struct student is created.
- The variable is passed to the display() function using display(s1); statement.

```
#include <stdio.h>
struct student {
  char name[50];
  int age;
};

// function prototype
void display(struct student s);
int main() {
  struct student s1;
  printf("Enter name: ");
  // read string input from the user until \n is entered
  // \n is discarded
  scanf("%[^\n]%*c", s1.name);
  printf("Enter age: ");
  scanf("%d", &s1.age);
  display(s1); // passing struct as an argument
  return 0;
}

void display(struct student s) {
  printf("\nDisplaying information\n");
  printf("Name: %s", s.name);
  printf("\nAge: %d", s.age);
}
```

```
Enter name: Bond
Enter age: 13

Displaying information
Name: Bond
Age: 13
```

Return struct from a function

- Here, the `getInformation()` function is called using `s = getInformation();` statement. The function returns a structure of type struct student. The returned structure is displayed from the `main()` function.
- Notice that, the return type of `getInformation()` is also `struct student`.

```c
#include <stdio.h>
struct student{
   char name[50];
   int age;
};
// function prototype
struct student getInformation();
int main(){
   struct student s;
   s = getInformation();
   printf("\nDisplaying information\n");
   printf("Name: %s", s.name);
   printf("\nRoll: %d", s.age);
   return 0;
}
struct student getInformation() {
  struct student s1;
  printf("Enter name: ");
  scanf ("%[^\n]%*c", s1.name);
  printf("Enter age: ");
  scanf("%d", &s1.age);
  return s1;
}
```

Passing struct by reference

- You can also pass structs by reference (in a similar way like you pass variables of built-in type by reference). We suggest you to read pass by reference tutorial before you proceed.
- During pass by reference, the memory addresses of struct variables are passed to the function.

```c
#include <stdio.h>
typedef struct Complex{
   float real;
   float imag;
} complex;
void addNumbers(complex c1, complex c2, complex *result);
int main(){
   complex c1, c2, result;
   printf("For first number,\n");
   printf("Enter real part: ");
   scanf("%f", &c1.real);
   printf("Enter imaginary part: ");
   scanf("%f", &c1.imag);
   printf("For second number, \n");
   printf("Enter real part: ");
   scanf("%f", &c2.real);
   printf("Enter imaginary part: ");
   scanf("%f", &c2.imag);
   addNumbers(c1, c2, &result);
   printf("\nresult.real = %.1f\n", result.real);
   printf("result.imag = %.1f", result.imag);
   return 0;
}
void addNumbers(complex c1, complex c2, complex *result) {
    result->real = c1.real + c2.real;
    result->imag = c1.imag + c2.imag;
}
```

```
For first number,
Enter real part:  1.1
Enter imaginary part:  -2.4
For second number,
Enter real part:  3.4
Enter imaginary part:  -3.2

result.real = 4.5
result.imag = -5.6
```

Passing struct by reference

- In the above program, three structure variables c1, c2 and the address of result is passed to the `addNumbers()` function. Here, result is passed by reference.
- When the result variable inside the `addNumbers()` is altered, the result variable inside the `main()` function is also altered accordingly.

LECTURE 6

Bit Vectors

Collections of Bools (Bit Vectors)

- Byte, word, ... can represent many Booleans

  One per bit, e.g., `00100101` = false, false, true, ..., true

- Bit-wise operations:

  Bit-wise AND: `00100101 & 10111100 == 00100100`

  Bit-wise OR: `00100101 | 10111100 == 10111101`

  Bit-wise NOT: `~ 00100101 == 11011010`

  Bit-wise XOR: `00100101 ^ 10111100 == 10011001`

Operations on Bit Vectors

```
const unsigned int  low_three_bits_mask = 0x7;
unsigned int        bit_vec = 0x15;
```

0…00 0111

0…01 0101

A *mask* indicates which bit positions we are interested in

Always use C's **`unsigned`** types for bit vectors

Selecting bits:

```
important_bits = bit_vec & low_three_bits_mask;
```

Result = ?

`0…00 0101 == 0…01 0101 & 0…00 0111`

Operations on Bit Vectors

```
const unsigned int  low_three_bits_mask = 0x7;
unsigned int        bit_vec = 0x15;
```

0…00 0111
0…01 0101

Setting bits:

```
bit_vec |= low_three_bits_mask;
```

Result = ?

0…01 0111 == 0…01 0101 | 0…00 0111

Operations on Bit Vectors

```
const unsigned int  low_three_bits_mask = 0x7;
unsigned int        bit_vec = 0x15;
```

0…00 0111
0…01 0101

Clearing bits:

```
bit_vec &= ~low_three_bits_mask;
```

Result = ?

0…01 0000 == 0…01 0101 & ~0…00 0111

Bit-field Structures

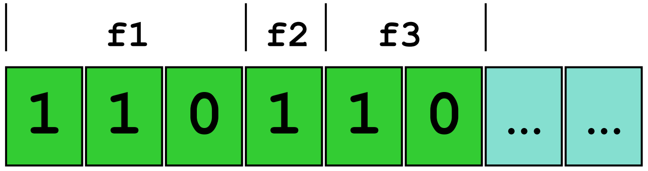
- Special syntax packs structure values more tightly
- Similar to bit vectors, but arguably easier to read
  - Nonetheless, bit vectors are more commonly used.

Padded to be an integral number of words

- Placement is compiler-specific.

```
struct Flags {
   int          f1:3;
   unsigned int f2:1;
   unsigned int f3:2;
} my_flags;

my_flags.f1 = -2;
my_flags.f2 = 1;
my_flags.f3 = 2;
```



LECTURE 7

Unions

Unions

Choices:

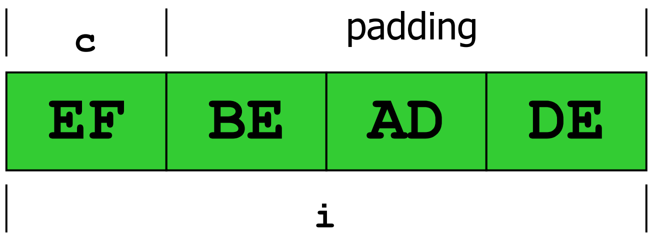
An element is

- an `int i` <u>or</u>
- a `char c`

`sizeof(union …)` = maximum of `sizeof(field)`

```
union AnElt {
    int   i;
    char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;
```



Unions

A union value doesn't "know" which case it contains

```
union AnElt {
   int   i;
   char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;

if (elt1 currently has a char) ...
```

?

How should your program keep track whether **elt1**, **elt2** hold an **int** or a **char**?

?

Basic answer: Another variable holds that info

Tagged Unions

Tag every value with its case

I.e., pair the type info together with the union

Implicit in Java, Scheme, ML, …

```
enum Union_Tag { IS_INT, IS_CHAR };
struct TaggedUnion {
   enum Union_Tag  tag;
   union {
      int   i;
      char  c;
   } data;
};
```

Enum must be external to struct, so constants are globally visible.

Struct field must be named.

LECTURE 8

Object-Oriented Programming in C

Overview

- Programming languages like C++ and Java have built-in support for OOP concepts. However, did you know that you don't need to use an OOP language in order to use OOP style and get some of the benefits of object-oriented programming?
- In this section, I will explain how we can bring some of the style of object-oriented programming to C, a language without built-in OOP support.

Simple, non-polymorphic types

Demo Program: oop/Point.h and oop/Point.cpp

```
class Point {
 public:
   Point(int x, int y);
   ~Point();
   int x() const;
   int y() const;
 private:
   const int x_;
   const int y_;
 };
```

```
Point::Point(int x, int y) : x_(x), y_(y) {}
Point::~Point() {}
int Point::x() const { return x_; }
int Point::y() const { return y_; }
```

C Object-Oriented Programming

Demo Program: oop2/Point.h and oop2/Point.cpp

- There are a number of important points to note about this translation.
- Firstly, we don't specify the full definition of "Point" in order to achieve encapsulation; we keep "x" and "y" effectively "private" by defining "Point" fully only in the source file.
- Secondly, we create functions that correspond to the constructor/destructor plus allocation/deallocation which replace "new" and "delete".
- Thirdly, all member functions get an explicit "self" parameter of the type being operated on (which replaces the implicit "this" parameter).

C Object-Oriented Programming

Demo Program: oop2/Point.h and oop2/Point.c

```c
#ifndef POINT_H

#define POINT_H

// Header

struct Point;  // forward declared for encapsulation

Point* Point__create(int x, int y);  // equivalent to "new Point(x, y)"

void Point__destroy(Point* self);  // equivalent to "delete point"

int Point__x(Point* self);  // equivalent to "point->x()"

int Point__y(Point* self);  // equivalent to "point->y()"

#endif
```

```c
// Source file
struct Point {
   int x;
   int y;
};
// Constructor (without allocation)
void Point__init(Point* self, int x, int y) {
  self->x = x;
  self->y = y;
 }
// Allocation + initialization (equivalent to "new Point(x, y)")
Point* Point__create(int x, int y) {
   Point* result = (Point*) malloc(sizeof(Point));
   Point__init(result, x, y);
   return result;
}
// Destructor (without deallocation)
void Point__reset(Point* self) {
}
// Destructor + deallocation (equivalent to "delete point")
void Point__destroy(Point* point) {
  if (point) {
    Point__reset(point);
    free(point);
  }
}
```

```
// Equivalent to "Point::x()" in C++ version
int Point__x(Point* self) {
  return self->x;
}
// Equivalent to "Point::y()" in C++ version
int Point__y(Point* point) {
  return self->y;
}
```

Polymorphism

- The patterns that apply to simple objects are also applied to other types of objects, but we add some enhancement to address concepts like polymorphism and inheritance.

Polymorphic types

Demo Program: oop3/*.h

- To create polymorphic types in C, we need to include additional type information in our objects, and we need some way of mapping from that type information to the customization that the type entails.

```cpp
// shape.h
class Shape {
public:
  virtual ~Shape() {}
  virtual const char* name() const = 0;
  virtual int sides() const  = 0;
};

// square.h
class Square : public Shape {
 public:
   Square(int x, int y, int width, int height)
      : x_(x), y_(y), width_(width), height_(height) {}
   virtual ~Square() {}

   const char* name() const override { return "Square"; }
   int sides()  const override { return 4; }

   int x() const { return x_; }
   int y() const { return y_; }
   int width() const { return width_; }
   int height() const { return height_; }

private:
   int x_;
   int y_;
   int width_;
   int height_;
};
```

```c
// shape.h
struct Shape;
struct ShapeType;

ShapeType* ShapeType__create(
   int buffer_size,
   const char* (*name)(Shape*),
   int (*sides)(Shape*),
   void (*destroy)(Shape*));

Shape* Shape__create(ShapeType* type);
ShapeType* Shape__type(Shape* self);
void* Shape__buffer(Shape* self);
int Shape__sides(Shape* self);
void Shape__destroy(Shape* shape);
```

Prototype for Shape

Demo Program: oop4/Shape.h

- In the above code, note that we created an extra object representing the type of the shape. This type information is how we perform dynamic dispatch (i.e. how we resolve virtual functions). You'll also note this funky "size" thing, which we use to allow a Shape to be allocated with additional space for a buffer, which we will use to store the data for a shape subclass.
- The basic idea is that for each type of shape (Square, Pentagon, Hexagon, etc.), there will be exactly one instance of the `ShapeType` struct. That is, every `Square` that we create will reference the exact same instance of `ShapeType` representing squares.

Polymorphism

- To summarize the code above, when inheritance/polymorphism is involved, it is necessary to encapsulate the polymorphic functions in a struct representing different derived types of the base type.
- Because the derived types may also add more data to the object, the allocation operation must allow the derived types to request additional space. It is also necessary to supply functions that can perform an up-cast and down-cast between the various data types.
- Additionally, virtual functions in C++ translate to functions that look up and dispatch to the type-supplied implementation of the given function.

Polymorphism

- Given the above, you might ask, why the extra layer of indirection of ShapeType? Why not simply store the function pointers representing the virtual function overrides directly on the Shape object (to be supplied in the create function of the various derived types).

- This is done for efficiency... combining the various virtual function overrides in a separate long-lived type object allows each instance to pay for just a single pointer field to support polymorphism; if this data were directly on the instances, themselves, then each instance would need as many additional fields as there are virtual functions.

LECTURE 9

Conclusion

Summary

- In this chapter, we covered struct, enum and union in C language.
- The struct and union are the heterogenous data type for C language. One with shared storage and one with distinct storage.
- The enum type is a data type for constants or user-defined iterating data type.