Complex data types

Lecture 8
CS 113 – Spring 2008

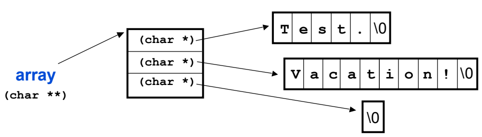
Announcements

- Assignment 2 due Sunday 11:59PM
  - Do <u>two</u> of the three problems
- Assignment 3 posted, due Monday 2/18 11:59PM
- Quiz next Wednesday in class

Arrays of strings

- Arrays of strings are just 2-D arrays of characters
  - Because strings are just arrays of characters

```
char *array2[] = {"Test.", "Vacation!", ""};
```



Example

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main()
{
  int str_count, j;
  char **str_array, buf[1024];

  printf("How many strings do you have? ");
  fgets(buf, 1024, stdin);
  str_count = atoi(buf);

  str_array = (char **) malloc(str_count * sizeof(char *));
  for(j=0; j<str_count; j++) {
    fgets(buf, 1024, stdin);
    str_array[j] = (char *) malloc(strlen(buf) + 1);
    strcpy(str_array[j], buf);
  }

  for(j=0; j<str_count; j++) printf("%s", str_array[j]);

  for(j=0; j<str_count; j++) free(str_array[j]);
  free(str_array);

  return 0;
}
```

Creating your own types

- C lets you name your own types using `typedef`
  - Example:

```
// create a new type called "scalar"
typedef double scalar;

scalar add(scalar a, scalar b) {
  return a + b;
}
```

- `typedef` syntax
  - As if you're declaring a variable of the type you want to redefine
  - Give the variable the name of the new type
  - Put `typedef` in front

typedef example

- Use typedef to create an *alias* for complex types

```
// create new types called "scalar" and "vector"
typedef double scalar;


scalar add_scalars(scalar a, scalar b) {
  return a + b;
}

add_vectors(scalar result[10], scalar a[10],
            scalar b[10]) {
  int j;

  for(j=0; j<10; j++)
    result[j] = a[j] + b[j];
}
```

typedef example

- Use typedef to create an *alias* for complex types

```
// create new types called "scalar" and "vector"
typedef double scalar;
typedef scalar vector[10];

scalar add_scalars(scalar a, scalar b) {
  return a + b;
}

add_vectors(vector result, vector a, vector b) {
  int j;

  for(j=0; j<10; j++)
    result[j] = a[j] + b[j];
}
```


Structures

- *Structures* aggregate variables of different types
  - Like *classes* in Java, but they can only store data
  - Defined with the keyword **struct**

```
// define new structure
struct Student {
  int id, year;
  char grade;
};

int main() {
  struct Student s;

  s.id = 10001;
  s.year = 2010;
  s.grade = 'B';

  printf("%d %d %c\n", s.id,
      s.year, s.grade);
}
```

struct Student

0	(int) id
4	(int) year
8	(char) grade

struct Student requires (approximately) 9 bytes

Structure syntax

- To define a structure

```
struct structure_name {
  /* list of variable declarations */
};
```

- To declare a variable of the new structure type

```
int main() {
  struct structure_name variable_name;
  /* code */
}
```

  - The **struct** keyword is required in both places

Structures and typedef

- To avoid the **struct** keyword, use a **typedef**
  - create alias for **struct Student_struct** called **Student**

```
// define new structure
struct Student_struct {
  int id, year;
  char grade;
};
typedef struct Student_struct Student;

int main() {
  Student s;

  s.id = 10001;
  s.year = 2010;
  s.grade = 'B';

  printf("%d %d %c\n", s.id, s.year, s.grade);
}
```


Another example: 2-D points

```
// define new structure
typedef struct {
  double x, y;
} Point;

double distance(Point p1, Point p2) {
  return (p1.x - p2.x) * (p1.x - p2.x) +
         (p1.y - p2.y) * (p1.y - p2.y);
}

int main() {
  Point a = { 12.0, 42.0 };
  Point b = { 15.0, 36.0 };

  printf("Distance: %lf\n", distance(a, b));
}
```


Structures in C

- C treats structures as if they were primitive data types
  - I.e. *not* like reference types in Java
  - Passing a structure as an argument to a function means that a temporary copy is made of the *entire structure*
  - Assignments do a *deep copy* of the entire structure

```
void reset(Point p) {
  p.x = p.y = 0;
}

int main() {
  Point a = { 12.0, 42.0 };
  Point b = a;

  reset(a);
  b.x = 0;
  printf("a: %d,%d\n", a.x, a.y);
  printf("b: %d,%d\n", b.x, b.y);
}
```

Structures and function calls

- Function calls with structure arguments can be costly
  - e.g. in this code, 10000+ bytes must be copied

```
typedef struct {
  char name[10000];
  int id;
} Person;

void print_person(Person p) {
  printf("%d %s\n", p.id, p.name);
}

int main() {
  Person a = { "Mike Smith", 1234 };
  print_person( a );
}
```


Pointers to structures

- Function calls with structure arguments can be costly
  - Solution: Use pointers to structures. Only the *address* of the structure is copied, not the entire structure.
- Use `&s` to get the address of a structure `s`
- Use `*p` to get the structure pointed to by `p`
- Use `(*p).field` to access a field
  - Or the cleaner syntax: `p->field`

Pointers to structures

- Function calls with structure arguments can be costly
  - Solution: Use pointers to structures. Only the *address* of the structure is copied, not the entire structure.

```
typedef struct {
  char name[10000];
  int id;
} Person;

void print_person(Person *p) {
  printf("%d %s\n", p->id, p->name);
}

int main() {
  Person a = { "Mike Smith", 1234 };
  print( &a );
}
```


Structures and the heap

- So far we've been storing structures on the stack
- Structures can also be stored on the heap
  - Call malloc() to request a memory region of the correct size
    - Use sizeof to determine the correct size
  - Cast the pointer returned by malloc to a pointer to the structure
  - Free the region when the structure is no longer needed

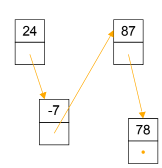
Structures and the heap

- Example

```
typedef struct {
  char name[10000];
  int id;
} Person;

int main() {
  Person *p;
  p = (Person *) malloc( sizeof(Person) );

  strcpy(p->name, "George Washington");
  p->id = 1776;

  free(p);
  return 0;
}
```


Practical example: Linked Lists

- A *linked list* is a very common data structure in CS
- The list is not (necessarily) stored in contiguous memory
  - Instead, each item is stored individually
  - Each item includes a reference to the next item in the list
  - The last item has a null reference
- Advantage:
  - List can grow and shrink dynamically
- Disadvantage:
  - No random access to the data
  - Memory overhead from next pointers



Linked lists in C

- ListCell structure holds one item and a next pointer

```
typedef struct ListCell_struct {
  int data;
  struct ListCell_struct *next;
} ListCell;
```

- Function to allocate a new ListCell on the heap
  - like a constructor in Java or C++

```
ListCell *make_ListCell(int data, ListCell *next) {
  ListCell *newcell;

  newcell = (ListCell *) malloc( sizeof(ListCell) );
  newcell->data = data;
  newcell->next = next;

  return newcell;
}
```


Linked lists in C

- Function to add a cell to the head of the list
  - returns a pointer to the new head of the list

```
ListCell *add_front(ListCell *head, int data) {
  return make_ListCell(data, head);
}
```

- Function to print the contents of a list

```
void print_list(ListCell *head) {
  for( ; head ; head = head->next)
    printf("%d\n", head->data);
}
```


Linked lists in C

- Function to destroy a ListCell
  - like a destructor for ListCell in C++

```
void destroy_ListCell(ListCell *cell) {
  free(cell);
}
```

- Function to destroy an entire list

```
void destroy_list(ListCell *head) {
  ListCell *next;
  for( ; head ; head = next) {
    next = head->next;
    destroy_ListCell(head);
  }
}
```


Linked lists in C

```
int main() {
  ListCell *head = NULL;    // list initially empty

  // Add 1, 2, 3, ... 10 to the list
  for(int j=0; j<10; j++)
    head = add_front(head, j);

  // Print the first element of the list
  printf("%d\n", head->data);

 // Print the fourth element of the list
  printf("%d\n", head->next->next->next->data);

  // Print the contents of the list
  print_list(head);

  // Free memory allocated to list
  destroy_list(head);
  return 0;
}
```

unions

- A *union* is a variable that stores objects of different types
  - But it can *only store one object at a time*
  - Like a structure, but all fields start at offset 0
- e.g. You have a floating point number
  - Sometimes you want to treat it as a float
  - Sometimes you want to treat it as an int
  - Sometimes you want to treat it as a char array

```
union new_float {
  float float_view;
  int int_view;
  char array_view[4];
};
```


union example

```
union new_float {
  float float_view;
  int int_view;
  char array_view[4];
};

int main() {
  union new_float f;

  f.float_view = 3.7;

  printf("%f %d %c %c %c %c\n", f.float_view, f.int_view,
         f.array_view[0], f.array_view[1], f.array_view[2],
         f.array_view[3]);

  return 0;
}
```

```
3.700000 406ccccd 40 6c cc cd
```