Introduction to C

Complex Types

Instructor: Yin Lou

02/02/2011

Recap: Numerical Types

- Data Types
  - int: machine-dependent
  - float: 32-bit (4 bytes)
  - double: 64-bit (8 bytes)
  - char: 8-bit (1 byte)

Recap: Numerical Types

- Data Types
  - int: machine-dependent
  - float: 32-bit (4 bytes)
  - double: 64-bit (8 bytes)
  - char: 8-bit (1 byte)
- Modifiers
  - short
  - long
  - signed
  - unsigned

Recap: Numerical Types

- Data Types
  - int: machine-dependent
  - float: 32-bit (4 bytes)
  - double: 64-bit (8 bytes)
  - char: 8-bit (1 byte)
- Modifiers
  - short
  - long
  - signed
  - unsigned
- Example: short int, unsigned char, long double, etc.

Complex Types

- Enumerations
  - User-defined weekday: sunday, monday, ...
- Structures
  - User-defined combinations of other types
- Unions
  - Same data, multiple interpretations
- Function pointers
- Arrays and pointers of the above

Enumerations

```
enum days {mon, tue, wed, thu, fri, sat, sun};
// Same as:
// #define mon 0
// #define tue 1
// ...
// #define sun 6

enum days {mon = 3, tue = 8, wed, thu, fri, sat, sun};
// Same as:
// #define mon 3
// #define tue 8
// ...
// #define sun 13
```

Enumerations

```
enum days day;
// Same as: int day;

for (day = mon; day <= sun; ++day)
{
    if (day == sun)
    {
        printf("Sun\n");
    }
    else
    {
        printf("day = %d\n", day);
    }
}
```

Enumerations

- Basically integers
- Can use in expressions like this
- Makes code easier to read
- Cannot get string equiv.
- Caution: day++ will always add 1 even if enum values aren't contiguous.

Creating Your Own Types

- C lets you name your own types using typedef

Creating Your Own Types

- C lets you name your own types using typedef

Example

```
typedef double scalar;

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

Creating Your Own Types

- C lets you name your own types using typedef

Example

```
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

More on typedef

```
typedef double scalar;
typedef scalar vector[10];

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

void add_vectors(vector result, vector, a, vector b)
{
    int i;
    for (i = 0; i < 10; ++i)
    {
        result[i] = a[i] + b[i];
    }
}
```

Structures

- Structures aggregate variables of different types
  - Defined with the keyword struct

Structures

- Structures aggregate variables of different types
  - Defined with the keyword struct

Example

```
struct student
{
    int id;
    int year;
    char grade;
};

void main()
{
    struct student s;

    s.id = 10001;
    s.year = 2011;
    s.grade = 'A';
}
```

Structure Syntax

- To define a structure

Structure Syntax

- To define a structure

Definition

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

Structure Syntax

- To define a structure

Definition

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

- To declare a variable of the new structure type

Structure Syntax

- To define a structure

Definition

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

- To declare a variable of the new structure type

Example

```
void main()
{
    struct structure_name variable_name;
}
```

Structure Syntax

- To define a structure

Definition

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

- To declare a variable of the new structure type

Example

```
void main()
{
    struct structure_name variable_name;
}
```

- The struct keyword is required in both places

Structures and typedef

- To avoid the struct keyword, use a typedef
  - Create alias for struct student called student_t

Structures and typedef

- To avoid the struct keyword, use a typedef
  - Create alias for struct student called student_t

Example

```
struct student
{
    int id;
    int year;
    char grade;
};
typedef struct student student_t;

void main()
{
    student_t s;

    s.id = 10001;
    s.year = 2011;
    s.grade = 'A';
}
```

Another Example: 2-D Points

```
// a shortcut to define structure
typedef struct
{
    double x;
    double 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);
}

void main()
{
    point a = {12.0, 42.0};
    point b = {15.0, 36.0};
}
```

Structures in C

- C treats structures as if they were primitive data types

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

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

Example

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

void main()
{
    point a = {12.0, 42.0};
    point b = a;

    reset(a);
    b.x = 0;
    printf("a: %lf, %lf\n", a.x, a.y); // a = 12.0, 42.0
    printf("b: %lf, %lf\n", b.x, b.y); // b = 0, 42.0
}
```

Structures and Function Calls

- Function calls with structure arguments can be expensive.

Structures and Function Calls

- Function calls with structure arguments can be expensive.

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);
}

void main()
{
    person a = {"Mike Smith", 1234};
    print_person(a);
}
```

Solution

- Use pointers. Only the *address* of the structure is copied, not the entire structure.

Solution

- Use pointers. Only the *address* of the structure is copied, not the entire structure.
- Use `&s` to get the address of a structure `s`

Solution

- Use pointers. 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 by `p`

Solution

- Use pointers. 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 by `p`
- Use (*p).field to access a field
  - Or a cleaner syntax:

    `p->field`

Pointers to Structures

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

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

void main()
{
    person a = {"Mike Smith", 1234};
    print_person(&a);
}
```

Structures and the Heap

- So far we've been storing structures on the stack

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

Example

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

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

void main()
{
    person *p;
    p = (person *) malloc(sizeof(person));
    strcpy(p->name, "Mike Smith");
    p->id = 1234;
    print_person(&a);
    free(p);
}
```

Unions

- A union is a variable that stores objects of different types

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

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

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

Example

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

Example

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

void 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]);
}
```

Function Pointers

```
int min(int a, int b);
int max(int a, int b);

int foo(int do_min)
{
    int (*func)(int, int); // declaring func. ptr

    if (do_min)
    {
        func = min;
    }
    else
    {
        func = max;
    }
    return func(10, 20); // indirect call
}
```

Function Pointers

- Points to a function
- Has a *-type of the function it points to