Structures

Data Types

- a set of values
- example from limits.h:

```
#define CHAR_MAX 127 /* max value for a char */
#define CHAR_MIN (-128) /* min value for a char */
#define INT_MAX 2147483647 /* max value for an int */
#define INT_MIN (-2147483647-1) /* min value for an int */
```

- operations on those values
Predefined Data Types

• insulate user from implementation
  – may be machine dependent
  – for example, floating point arithmetic
• manipulated by predefined operations
  – implementation of operations also hidden
• use is completely determined by predetermined semantics
• implicitly well-known (integer arithmetic)
• explicitly stated
User-defined Types

- built from
  - built-in types (int, float, char,...)
  - type constructors (array, struct)
  - and access operations (field selection)
- completely visible to user
- operations provided as user-defined fns
  - not tied to data type
  - visible implementation
Abstract Data Types (ADT)

- set of data
- set of operations on data
- independent of implementation
- may support a variety of implementations
- example: a list ADT may have ops
  front, tail, insert, isempty
Struct

- C/C++ mechanism for building user-defined types
- A way to connect physically separate but logically defined data
- Handles heterogeneous data types
- Example: student data consisting of id, name, major code and gpa
Defining structs (defining the data type)

const int MAXNAME = 50;
struct StInfo
{
    int id;
    char name[MAXNAME];
    double majorcode;
    double gpa;
};

Declaring instances of a struct

• example

    StInfo alice, bob, chuck;

Initializing structs
• example

    StInfo alice, bob={654321,"Bob Brown", 422.20, 2.97}, chuck;

• Omitting data is OK – only at the end:

    StInfo bob={654321,"Bob Brown", 422.20 }; \ OK
    StInfo chuck={235416, , 421.21,}; \ not OK

Accessing members of a struct

• use · operator

• assignment

    alice.gpa = 3.25;
    alice.id = 123456;

• function call

    strcpy(alice.name, "Alice Smith");
• relational ops and output

```cpp
if (bob.majorcode == 422.20)
{
    cout << bob.name << " is a CS major " << endl;
}

cout << alice.name << endl;
```

• need to access built-in types

• access members but not whole structs

```cpp
cout << alice << endl; // illegal

if (alice == bob) ... // illegal
```
Arrays of Structs

- Declaration

```cpp
int CLASSSIZE = 30;
StInfo classlist[CLASSSIZE];
```

- conceptually

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>majorcode</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- array traversal

```cpp
for (int i=0; i < CLASSSIZE; i++)
{
    cout << classlist[i].id << " "
        << classlist[i].name << " "
        << classlist[i].majorcode << " "
        << classlist[i].gpa << endl;
}
```
• Initializing an Array of Structs

  – Example

  ```c
  const int CLASSSIZE = 30;

  StInfo classlist2[CLASSSIZE] =
  {
   {12345, "Annie Jones", 422.0, 3.92},
   {543216, "Bart Doe", 422.10, 3.01}
  };

  – Note: must use `const int`
  ```
● Nesting Structs

```c
struct Addr
{
    char str[MAXLEN];
    char city[MAXLEN];
    char state[STCODE];
    int zip;
};

struct StInfo2
{
    int id;
    char name[MAXNAME];
    Addr loc;
    double majorcode;
    double gpa;
};

struct StInfo2 annie;
annie.majorcode = 431.10;
strcpy(annie.loc.city,"San Marcos");
```
• Structs as Function Arguments
  – Using Struct Members
    ```cpp
    void gpaReport(int id, double gpa)
    {
      cout << id << " " << gpa << endl;
    }
    
gpaReport(alice.id, alice.gpa);
    ```

    Reference on file organization:
    http://www.gamedev.net/reference/articles/article1798.asp

  – Using Structs
    ```cpp
    void gpaReport2(StInfo st)
    {
      cout << st.id << " " << st.gpa << endl;
    }
    ```
gpaReport2(alice);

• Structs Can Be Reference Parameters

```c
void gpaReport3 (StInfo &st, double newmajorcode) {
  st.majorcode = newmajorcode;
}
```

gpaReport3(alice, 432.10);

• Structs Can Be const Reference Parameters
  – reference saves copying
  – const protects data
• Structs Can Be Return Values

```cpp
StInfo getStInfo ()
{
    StInfo st;
    cin >> st.id;
    cin >> st.name;
    cin >> st.majorcode;
    cin >> st.gpa;
    return st;
}

bob = getStInfo();
```
• Pointers to Structs
  
  – Example

  ```c
  StInfo *p;
  
p = &alice;  // p points to a struct
  
p->gpa = 2.0;  // accesses member of a struct
  ```

  – precedence of operators, high to low

<table>
<thead>
<tr>
<th>·</th>
<th>direct selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>indirect selection</td>
</tr>
<tr>
<td>*</td>
<td>indirection or dereference</td>
</tr>
</tbody>
</table>

  Consider these:

  ```c
  *p.gpa = 2.0;  // NO parses as *(p.gpa)
  (*p).gpa = 2.0;  // OK accesses a member of a struct
  ```
• Pointers to Structs as Function Parameters

```c
void updatemajorcode2(StInfo *st, double newmajorcode)
{
    st -> majorcode = newmajorcode;
}

updatemajorcode2(&alice, 543.21);
```

• Dynamic Memory Allocation for Structs

```c
StInfo * st;
st = new StInfo;
st->id = 246810;
strcpy(st->name, "John Doe");

int ii;
StInfo *classptr;
classptr = new StInfo[30];
classptr[ii].id = 234678; // it is an array
```
• Pointers as Members of Structs
  – Pay attention to operator precedence
  – Example: both expressions access grades[0]

    const int MAXNAME = 50;
    struct StInfo
    {
        int id;
        char name[MAXNAME];
        double majorcode;
        double gpa;
        int *grades; // dynamically allocated array
    };

    StInfo alice;
    StInfo * st;

    cout << *st->grades; // *(st->grades)
    cout << *alice.grades; // *(alice.grades)