Enumerations

Symbolic constants

- An enumeration is a set of named integers
- Provide mnemonic names for "magic" numbers
  - Passing and returning encoded values
  - Creating bit values (masks), which may be ORed & ANDed together

```c
enum tag { enumeration_list } variable_list;
```
- `tag` is optional and follows the rules for creating a legal identifier
- `variable_list` is optional and often omitted; if present, it defines variables of type `tag`
- `enumeration_list` is a comma separated list if legal identifiers
  - the first identifier is assigned the value 0
  - the value of one or more identifiers may be specified
  - unless specified, the value of an identifier is always 1 greater than the last

Enum Examples

"Magic numbers" and bit-patterns

```c
enum commands { // commands typed at keyboard
e  Unknown, // unknown command entered
e  Quit, // close files, quite database
e  Search, // search for a person
e  Input, // input a new person
e  Import, // import data from another file
e  Dump, // dump database in ascii/text
e  Help, // show valid commands list
}
```

```c
enum { ured = 1, uwrite = 2, uexe = 4, gread = 8, gwrite = 16,
gexe = 32, oread = 64, owrite = 128, oexe = 256
};
```

typedef Statements

Creating type aliases

- A typedef statement creates an alias or new name for an existing type (i.e., does not create a new type)
  - A "shorthand" notation
  - More mnemonic or self documenting
  - Improve portability by creating machine-dependent types (see <sys/types.h>)

- General format: `typedef old_name new_name;`
  - `typedef unsigned long ulong;`
  - `typedef unsigned int size_t;`

```c
ulong counter;
size_t length;
```

Enumerations Examples

Continued

```c
commands command = getcommand();
switch(command)
{ case e_Quit : exit(0);
case e_Search : search(personFile, indexFile);
break;
case e_Input : input();
break;
case e_Import : import(personFile, indexFile);
break;
case e_Dump : dump(personFile);
break;
case e_Help  : help();
break;
default : cerr << "UNKNOWN command
";
break;
}
```

Structures

User-created aggregate types

- Collection of variables referenced with one name
- A structure declaration is a blueprint, or "cookie cutter" (i.e., does not allocate memory); often put in header files
- A structure declaration is a blueprint, or "cookie cutter" (i.e., does not allocate memory); often put in header files

- General format: `struct tag { variable declarations; };
```
- Creates a new type specifier (i.e., a new data type)
- Variable declarations may include any type except the struct-type being created
- Each element or variable is called a member or field
**Structs vs Database Tables**

Members are equivalent to columns

<table>
<thead>
<tr>
<th>modelnumber</th>
<th>partnumber</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>687</td>
<td>10.57</td>
</tr>
<tr>
<td>9673</td>
<td>108</td>
<td>57.03</td>
</tr>
<tr>
<td>4957</td>
<td>892</td>
<td>3.99</td>
</tr>
</tbody>
</table>

```cpp
struct part {
    int modelnumber;
    int partnumber;
    float cost;
};
```

```cpp
part p1 = {1234, 687, 10.57};
p2;
p3 = new part;
cin >> p2.modelnumber;
cin >> p2.partnumber;
cin >> p2.cost;
cin >> p3->modelnumber;
cin >> p3->partnumber;
cin >> p3->cost;
```

**Member Access Operators**

Accessing the individual data contained in a struct

- If the left hand side (LHS) operand is a struct object, use the . (the period, pronounced "dot")
- If the LHS operand is a pointer to a struct, use the -> (formed by the minus and greater than symbols, pronounced "arrow")
- Members are legal anywhere a variable of that type is legal (i.e., they can be either l-values or r-values)
- Members (i.e., names) are in a unique scope

```cpp
when.year = 1998;
int mon = when.mon;
datePtr->day = 14;
int year = datePtr->year;
```

**Manipulating structs**

Miscellaneous syntax

- Finding the address: &today;
- Finding the size (in bytes): sizeof(date);
- Allocating structs dynamically: date* dateVar;
- Allocating new date;
- Deallocating dynamic memory or variables: delete dateVar;

**Structure Function Arguments**

Passing structs by value and by address

```cpp
void printDate1(date D)
{
    cout << D.year << endl;
    cout << D.mon << endl;
    cout << D.day << endl;
}
```

```cpp
date newYears = {1, 1, 2004};
printDate1(newYears);
```

```cpp
void printDate2(date* D)
{
    cout << D->year << endl;
    cout << D->mon << endl;
    cout << D->day << endl;
}
```

```cpp
date newYears = {1, 1, 1998};
printDate2(&newYears);
```
Structure Return Values

Return by value

date getDate(void) {
    date D;
    cout << "enter a year: ";
    cin >> D.year;
    cout << "enter a month: ";
    cin >> D.mon;
    cout << "enter a day: ";
    cin >> D.day;
    return D;
}

now = getDate();

date now;
now = getDate();

Nested structs

struct within structs

- structs may contain structs
  - range R;
  - R.Start.year = 1776;
- structs cannot contain themselves
- structs can contain pointers to themselves

struct range {
    date Start;
    date End;
};

struct date {
    int year, mon, day;
    // error
};

struct ListNode {
    char* Name;
    ListNode* Next;
};

unions

A "schizophrenic" type

- Look like structs but use the keyword union
- Can contain data of multiple types but only one type at a time
  - structs reserve memory for all fields
  - unions reserve memory for the largest field (memory is shared)
  - structs and unions are often used together
- May be used to create variables capable of storing more than one type of data
- Fields are accessed with either . (dot) or -> (arrow)
- May be used for type conversions (but may not be portable)
  - data stored in one union element (as a bit pattern)
  - data read out of another union element (as a bit pattern)

struct vs union

Memory layout and consumption

struct demo {
    char c;
    int i;
    double d;
};

union constant {
    char c;
    int i;
    double d;
};

Bit Fields

Small, odd-sized data

- Based on structs (i.e., are an extension of the struct syntax)
- Used to access non-standard sized data (i.e., not 1, 2, 4, or 8 bytes); usually less than 1 byte (8 bits)
- Each field is followed by a size, measured in bits
- Example:

```c
struct tag {
    unsigned int field1 : 3;
    unsigned int field2 : 7;
    unsigned int field3 : 6;
};
```

Bit Field Example

From stat.cpp

```c
struct modes {
    unsigned int type : 7;
    unsigned int user : 3;
    unsigned int group : 3;
    unsigned int others : 3;
};

union short_to_modes {
    unsigned short file_mode;
    struct modes conv;
};
```