Review of / Introduction to Structures

1 Review/Introduction to Structures

`struct`

- Group related items together.
- Machine/compiler dependencies exist (structure alignment).
- Unnecessary if classes are used.

Accessing Fields in a `struct` variable

Use a dot (`.`) for *static* structure variables.

Use an arrow (`->`) for *dynamic* structure variables.

These are illustrated in the example that follows.

Declare a `struct` to represent a fraction

```
struct fraction
{
   int  numer;
   int  denom;
};

typedef struct fraction Fraction;
```

Note:

- curly braces and semi-colons!
- `typedef` defines a new data type.

Example: Fraction Program

```
#include <iostream>

using namespace std;

struct fraction
{
   int  numer;
   int  denom;
};

typedef struct fraction Fraction;

void SetFraction( Fraction& f, int n, int d );
void PrintFraction( Fraction f );
```

Note: The `&` in the prototype for `SetFraction()` means *pass by reference.*

```
int main()
{
   Fraction f;

   f.numer = 1;
   f.denom = 2;
   cout << "one half: ";
   PrintFraction( f );
   cout << endl;

   SetFraction( f, 1, 3 );
   cout << "one third: ";
   PrintFraction( f );
   cout << endl;

   return 0;
}
```

Output

```
one half: 1 / 2
one third: 1 / 3
```

What do `SetFraction()` and `PrintFraction()` look like?

```
/* SetFraction:  Initialize a fraction  */
void SetFraction( Fraction& f, int n, int d )
{
   f.numer = n;
   f.denom = d;
}

/* PrintFraction:  Print a fraction  */
void PrintFraction( Fraction f )
{
   cout << f.numer << " / " << f.denom;
}
```

Dynamically Allocated Fractions

Need to use pointers for dynamically allocated fractions. This requires a few minor modifications to the code.

```
typedef struct fraction Fraction;
typedef struct fraction * FractionPtr;
```

These statements define two new types: `Fraction` and `FractionPtr`. This makes coding a bit easier.

Prototypes for Dynamically Allocated Fractions

```
typedef struct fraction Fraction;
typedef struct fraction * FractionPtr;

      // Static
void SetFraction( Fraction& f, int n, int d );
void PrintFraction( Fraction f );

      // Dynamic
void SetFraction( FractionPtr & f, int n, int d );
//void SetFraction( (Fraction*) & f, int n, int d );  // fail!
void PrintFraction( FractionPtr f );
//void PrintFraction( Fraction *f );  // works
```

```
void TestDynamicFractions()
{
   cout << "\nTestDynamicFractions():" << endl;

   FractionPtr f = new Fraction;  // must allocate!

   f->numer = 1;
   f->denom = 2;
   cout << "one half: ";
   PrintFraction( f );
   cout << endl;

   SetFraction( f, 1, 3 );
   cout << "one third: ";
   PrintFraction( f );
   cout << endl;
}
```

```
/* SetFraction:  Initialize a fraction  */
void SetFraction( FractionPtr & f, int n, int d )
//void SetFraction( (Fraction*) & f, int n, int d )
{
   f->numer = n;
   f->denom = d;
}

/* PrintFraction:  Print a fraction  */
void PrintFraction( FractionPtr f )
//void PrintFraction( Fraction *f )
{
   cout << f->numer << " / " << f->denom;
}
```

Morse Code Translation

Morse code translation problem.

a	`.-`
b	`-...`
c	`-.-.`
d	`-..`
e	`.`
	⋮
r	`.-.`
s	`...`
t	`-`

Note that frequently used letters are encoded with fewer *dots* and *dashes*. This is by design!

Let's examine a similar problem.

Integer to Character Conversion Problem

Convert an integer (0–9) to a character and print it.

This simple example illustrates a solution to a problem that is similar to the Morse code translation problem.

Why is it important to have a clear description of the problem we want to solve?

1.0.1 Possible Solutions

- ASCII table
- `if-else` (or `switch`) (long)
- Character arithmetic (`'0' + n`)
- "Lookup" table (array, or array of some `struct`)

1.0.2 `if()-else if()` Solution

```
char IntToChar( int n )
{
   char cRetVal;

   if( n == 0 )
      cRetVal = '0';
   else if( n == 1 )
      cRetVal = '1';
   ...
   else if( n == 9 )
      cRetVal = '9';

   return cRetVal;
}
```

Thoughts

- How long? (About 20 lines.)
- Easy to program (a bit tedious).

1.0.3 Character Arithmetic Solution

Recall that we can do character arithmetic, but we must use some caution when doing so.

```
return  '0' + n;
```

1.0.4 Character Array Solution

Use an array of characters that match the index (number).

```
char cTable[] = {'0', '1', ..., '9'};

return cTable[n];
```

Reflection

- Are these solutions better?
- Why?

Note: No error checking is included in any of these solutions! All solutions are very specific to this problem.

Can we develop a more general solution that we could possibly extend to solving the Morse code translation problem?

1.0.5 Table Solution

Use a table of values (variation of the array solution). The table contains an integer value and the character equivalent. Do they need to be organized in any particular order?

```
struct IC_Entry {
   int  iVal;
   char cVal;
};

const int MAX_IC_TABLE_SIZE = 10;
struct IC_Entry IC_Table[MAX_IC_TABLE_SIZE];
```

Table Initialization

```
IC_Table[0].iVal = 0;
IC_Table[0].cVal = '0';
IC_Table[1].iVal = 1;
IC_Table[1].cVal = '1';
     ...
IC_Table[9].iVal = 9;
IC_Table[9].cVal = '9';
```

Tedious programming—Copy/Paste or lots of typing! Errors?

What if the table was bigger?

Other ways to initialize/create table?

- Programmatically (loop)
- Program to generate the code.

Table Lookup

```
char IntToChar( int i )
{
   return table[i].cVal;
}
```

Morse Code Thoughts

Find value in table (search)?

- Comparison

Translation algorithm requires match/comparison.

- Easy for basic data types more difficult for aggregate types.

- Reconsider the number to character example:

```
for  i = 0, i < N
   if  table[i].iVal == n
      return table[i].cVal
```

What about *strings?*