Data Types

- Definition: *Data Type*

  A data type is a collection of values and the definition of one or more operations on those values.

- Base Data Types

  All the values of the type are ordered and atomic.

  C++: `short int, int, long int, float, double, long double, char`

- User-defined Types

  All the values of the type are ordered and atomic.

  C++: determined by the user

- Aggregate Data Types

  Complex combinations of base data types and user-defined types.

  C++: arrays, structures

Enumerated Types

Enumerated types list all the valid values for a type.

```
enum Season     { WINTER, SPRING, SUMMER, FALL};
enum Hemisphere { NORTH, SOUTH, EAST, WEST};
Season      period;
Hemisphere  region;

period = SUMMER;
region = EAST;
```

Enumerated types are, by definition, integer types whose values start with 0, unless otherwise specified, and are incremented by 1.

```
enum Months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};

enum Coin {PENNY = 1, NICKEL = 5, DIME = 10};

enum Boolean {FALSE = 0, TRUE = 1};  // ! enum Boolean {TRUE , FALSE };
```

Enumerated Types (cont)

They provide a mechanism for abstraction of real world entities.

Enumerated type variables do not contain the character string of the value. The internal representation uses `int` values; it is bad form to rely on those values.

Values of a given enumerated type must be unique.

Identifier names of enumerated types must be unique, cannot be used in other enumerated types.

Enumerated types cannot be read or (usefully) written directly.

Enumerated data types make the program more readable by providing a way to incorporate meaningful labels (thus easier to debug) and they help in self-documenting the program.

Enumerated Types Example

```
enum ParityType { EVEN, ODD };            // file-scoped type definition

ParityType FindParity(int N);

void main() {
   int k;
   ParityType kParity;

   cout << "Please enter an integer: ";   // get an integer
   cin  >> k;

   kParity = FindParity(k);               // determine its parity

   if (kParity == EVEN)                   // can't usefully print
      cout << "That was even." << endl;   // the value of kParity
   else                                   // directly
      cout << "Hmmm... that was odd." << endl;
}

ParityType FindParity(int N) {

   if (N % 2 == 0)
      return EVEN;
   else
      return ODD;
}
```

Type Coercion

C++ performs implicit type conversion whenever values of different types are used in:

- arithmetic and relational expressions
- assignment operators
- parameter passage
- return of a function value from a value-returning function

Promotion (widening): conversion from a "lower" type to a "higher" type,
e.g. _____

Demotion (narrowing): conversion from a "higher" type to a "lower" type,
e.g. _____

Compound Operators

Compound operators: Allow arithmetic operations and assignment operation in one statement, e.g.

```
y += x;        means _____________________
y -= x;        means _____________________
y *= x;        means _____________________

y /= x;        means _____________________
```

The compound operators provide a convenient contraction that saves typing, but may reduce code readability.

Binary Shift Operators

Shift operators are only for integer operands.

The shift operator shifts the bits of the operand in the indicated direction, inserting zeros at the opposite end.

The effect of a shift to the left (<<) is multiplication by a power of 2

The effect of a shift to the right (>>) is division by a power of 2.

Suppose `someInt = 32`, then

```
result = (someInt << 2);   // result = 32 * 2^2 = 128
                           // shifts 0's from the right

result = (someInt >> 3);   // result = 32 / 2^3 = 4
                           // shifts 0's from the left
                           // machine dependent
```

Bitwise Operators

Given:

```
int someInt1 = 5 , // someInt1 == “0000 0000 .... 0000 0101”
    someInt2 = 3 ; // someInt2 == “0000 0000 .... 0000 0011”
```

Bitwise AND: `someInt1 & someInt2 == _____`

Bitwise OR: `someInt1 | someInt2 == _____`

Bitwise exclusive OR: `someInt1 ^ someInt2 == _____`

Complement: `~someInt1 == _____`

Cast Operators

Cast operators allow the explicit changing of the data type of expressions, e.g.

```
ratio = float (a) /b;       // functional notation

ratio = (float) a / b;     // prefix notation -use
                           // typecasting expressions
```

Cast operators are available for any built-in data type. It is generally better to use an explicit typecast rather than rely on the invisible coercion mechanisms in C++. For instance, given the declarations

```
int m;
double x = 5.0, y = 2.0;
```

write `m = int(x / y)` rather than `m = x / y`

The former makes the automatic truncation explicit, and indicates the programmer is aware that the truncation will occur.

Other Operators

`sizeof` operator: yields the size in bytes of its operand, e.g.

```
sizeof(someInt)
```

returns the size in bytes of the storage allocated for `someInt`, which would be 4 on most contemporary systems.

Conditional ternary operator: `expr1 ? expr2 : expr3;`

is a contracted form of an if..else statement with assignments. It provides brevity at the expense of code readability.

```
Nparity = (N % 2 == 0) ? EVEN : ODD;
```

```
if (N % 2 == 0)
   Nparity = EVEN;
else
   Nparity = ODD;
```

Typedef

The `typedef` statement is used to allow the programmer to custom tailor his/her types by using pre-existing types in a way that makes her/his types more meaningful.

```
typedef predefined_type  newDefinedType ;
```

E.g.:

```
const int MaxNameLength = 30;

typedef char Name[MaxNameLength];

Name myName = “Fred Flintstone”;

Name yourName = “Mud”;

cout << myName << endl;
```

Typedefs allow the programmer to map existing data types to a problem-specific concept.

Good use of typedefs can improve data organization and program efficiency.

Correct application of typedefs aid in the production of self-documenting code.