

data type: a collection of values and the definition of one or more operations that can be performed on those values

C++ includes a variety of built-in or base data types:

`short`, `int`, `long`, `float`, `double`, `char`, etc.

The values are ordered and atomic.

C++ supports several mechanisms for aggregate data types: arrays, structures, classes.

These allow complex combinations of other types as single entities.

C++ also supports other mechanisms that allow programmers to define their own custom data types: `enum` types and `typedefs`.

An enumerated type is defined by giving a name for the type and then giving a list of labels, which are the only values that a variable of that type is allowed to have.

Enumerated types allow the creation of specialized types that support the use of meaningful labels within a program. They promote code readability with very little overhead in memory or runtime cost.

```
enum Month      {JAN, FEB, MAR, APR, MAY, JUN,
                JUL, AUG, SEP, OCT, NOV, DEC};
enum Season     {WINTER, SPRING, SUMMER, FALL};
enum Hemisphere {NORTH, SOUTH, EAST, WEST};

Month          Mon;
Season         Period;
Hemisphere     Region;

. . .

if (Mon == JAN && Region == NORTH)
    Period = WINTER;
```

An enumerated type is allowed to have up to 256 values and a variable of an enumerated type will occupy one byte of memory.

It is an error for the same label to be listed as a value for two different enumerated types.

Enumerated types provide a mechanism for the abstraction of real world entities.

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

Since the labels are logically constant values it is common to express them using all capital letters, just as for named constants.

Enumerated type values cannot be read or (usefully) written directly.

Enumerated type variables and values can be used in relational comparisons, in assignments and in switch statements as selectors and case labels.

Enumerated type variables can be passed as parameters and used as the return value of a function.

Good use of 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.

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

ParityType FindParity(int N);

int 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;

    return 0;
}

ParityType FindParity(int N) {

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

Since an enumerated type is represented in hardware by integer values, enum variables may be used as switch selectors and as array indices:

```
string parityToString(ParityType Parity) {  
  
    const int NUMPARITYVALUES = 2;  
    const string parityLabel[NUMPARITYVALUES] = {"even", "odd"};  
  
    if ( int(Parity) >= NUMPARITYVALUES ) {  
        return "unknown";  
    }  
    return parityLabel[Parity];  
}
```

User input strings can be represented internally using an enumerated type:

```
enum editCommand {INSERT, DELETE, MARKBEGIN, MARKEND, . . . , EXIT,  
                 EDITUNKNOWN};  
  
. . .  
editCommand getCommand(ifstream& In) {  
  
    string userEntry;  
    getline(In, userEntry, '\\t');  
    if ( userEntry == "insert" ) return INSERT;  
    if ( userEntry == "delete" ) return DELETE;  
    . . .  
    return EDITUNKNOWN;  
}
```

The typedef statement allows a programmer to define custom types by creating an "alias" for an existing type:

```
typedef <predefined type> <alias>;
```

For example:

```
typedef string PlaceName;  
typedef int Vector3D[3];  
  
void Reverse(Vector3D X) {  
    X[0] = -X[0];  
    X[1] = -X[1];  
    X[2] = -X[2];  
}
```

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

Good use of typedefs can improve data organization.

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

... but no useful additional type-checking is enabled.