Data Structures
in C++

Chapter 1

Tim Budd

Oregon State University
Corvallis, Oregon
USA
Language Features

The purpose of this chapter is to *quickly* review all those features of C++ that you should *already* have encountered and be familiar with.

Even if you have learned programming in another language (Pascal, for example) you should be able to quickly get up to speed with the features described here.
Comments

There are two forms of comments in C++

    // from slashes to end of line

    /*
     * comments that span
     * multiple lines
     */

Comments should be used extensively for documentation.
Constants

There are various types of constants:

- integer – 1, 12, –37
- octal integers – 014
- hexadecimal integers 0XFF 0XC
- floating point – 3.14159 2.7e14
- character – ’a’ ’\n’
- string – ”abc”

Suffixes can be applied to integer constants (U for unsigned, L for long)

Several other special backslash characters
Variables, Types, Values and Declarations

A variable is a named location that can hold values of a certain type.

Variables are created using a declaration statement, which also describes the associated type.

```c
int a, b, c // declare three integer variables
```

Declarations can be combined with initialization:

```c
double pi = 3.1415926;
```
Fundamental Data Types

The fundamental data types:

- integer – int
- floating point – double, float
- character – char

Modifiers that can be used with fundamental types

- signed, unsigned – positive and negative, or positive only
- short, long – (possibly) shorter or longer than standard
More Data Types

Boolean (bool) variables are true/false.

Enumerated values are defined by providing an explicit range

```cpp
enum months {January, February, March, April, May, June, July,
             August, September, October, November, December};

months workingMonth, vacationMonth;
months summerMonth = August;
```
Variables and Assignment

Variables are modified by assignment statements, which assign an expression to a variable.

```c
double f, c;  // Fahrenheit and Celsius temperature

  c = 43;
  f = (c * 9.0) / 5.0 + 32;
```

Binary operators can be combined with assignment:

```c
i += 5;
```

has the same meaning as the statement:

```c
i = i + 5;
```

Other short-hand notations:

```c
i++
```

has the effect of incrementing the variable `i` by one.
# Lots of Operators

<table>
<thead>
<tr>
<th><strong>Unary Operators</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>increment, decrement</td>
<td>i++, ++i, i--, --i</td>
</tr>
<tr>
<td>negation</td>
<td>( -i )</td>
</tr>
<tr>
<td>bit-wise inverse</td>
<td>( \sim i )</td>
</tr>
<tr>
<td><strong>Arithmetic Operations</strong></td>
<td></td>
</tr>
<tr>
<td>addition, subtraction</td>
<td>( a+b )</td>
</tr>
<tr>
<td>multiplication, division</td>
<td>( a \times b )</td>
</tr>
<tr>
<td>remainder after division</td>
<td>( a \mod b )</td>
</tr>
<tr>
<td><strong>Shift Operations (also stream I/O)</strong></td>
<td></td>
</tr>
<tr>
<td>left shift (also stream output)</td>
<td>( a &lt;&lt; b )</td>
</tr>
<tr>
<td>right shift (also stream input)</td>
<td>( a &gt;&gt; b )</td>
</tr>
<tr>
<td><strong>Relational Operations</strong></td>
<td></td>
</tr>
<tr>
<td>less than, less than or equal</td>
<td>( &lt; &lt;= )</td>
</tr>
<tr>
<td>equal, not equal</td>
<td>( == ! = )</td>
</tr>
<tr>
<td>greater than, greater than or equal</td>
<td>( &gt; &gt;= )</td>
</tr>
<tr>
<td><strong>Logical Operations</strong></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>( x &amp; &amp; y )</td>
</tr>
<tr>
<td>or</td>
<td>( x</td>
</tr>
<tr>
<td>logical negation</td>
<td>( ! i )</td>
</tr>
<tr>
<td><strong>Miscellaneous Operations</strong></td>
<td></td>
</tr>
<tr>
<td>function call</td>
<td>( f(a,b,c) )</td>
</tr>
<tr>
<td>conditional expression</td>
<td>( c ? a : b )</td>
</tr>
</tbody>
</table>
Stream I/O

The left and right shift operators are given different meanings when used with stream values. The most common stream is associated with “console input and output”.

```cpp
cout << "the Fahrenheit equivalent of " << c << 
" is " << f << "\n";

cin >> c; // get a new value of c
cout << "the Fahrenheit equivalent of " << c << 
" is " << f << "\n";
```

Operator »» works by side effect, changing the right hand expression. Result can be converted into a boolean, to test if input was successful.

```cpp
int sum = 0;
int value;
while (cin >> value) {
    sum += value;
}
cout << "sum is " << sum << "\n";
```
Pointers

A pointer is a variable that maintains the address of another location in memory.

Pointers can be used in the following ways:

- Pointers can be subscripted, (works best if they point to an array, but isn’t checked).
- Pointers can be dereferenced, using the * operator.
- Can combine dereference and field access, using -> operator.
- Can use addition, p+i is address of p[i].
Conditional Statements

Normal sequential control can be modified using a conditional statement:

```cpp
month aMonth;
...
if ((aMonth >= June) && (aMonth <= August))
    isSummer = true;
else
    isSummer = false;
```

Else part is optional.
Switch Statements

Switch statements can select one of many alternatives:

```c
switch (aMonth) {
    case January:
        highTemp = 20;
        lowTemp = 0;
        break;
    case February:
        highTemp = 30;
        lowTemp = 10;
        break;
    ...
    case July:
        highTemp = 120;
        lowTemp = 50;
        break;
    default:
        highTemp = 60;
        lowTemp = 20;
}
```
Loops

Loops are used to execute statements repeatedly until a condition is satisfied.

```cpp
int c = 0;
while (c <= 100) {
    cout << "Celsius " << c << " is Fahrenheit " << ((9.0 * c) / 5.0 + 32) << "\n";
    c += 10;
}
```
For statements

For statements combine in one statement initialization, termination test, and update.

```cpp
for (c = 0; c <= 100; c += 10) {
    cout << "Celsius " << c << " is Fahrenheit " <<
         ((9.0 * c) / 5.0 + 32) << "\n";
}
```

Declaration of new variables can be combined with loop.

```cpp
for (int i = 0; i < 12; i++) {
    cout << "i: " << i << " i squared " << i*i << "\n";
}
```
Array

An array is a fixed sized collection of similarly-typed values. Array elements are accessed using subscripts, range is zero to one less than array size.

```cpp
// declare an array of twelve integer values
int Temperatures[12];
// now assign all values
Temperatures[0] = 0;
Temperatures[1] = 10;
...
```

Arrays can be initialized:

```cpp
string MonthNames[12] = {
    "January", "February",
    "March", "April", "May", "June",
    "July", "August", "September", "October",
    "November", "December"};
```
Multidimensional Arrays

Arrays of more than one dimension can be created by giving the extent along each axis.

```cpp
double matrix[10][20];
```

Creates a double precision array of ten rows and twenty columns. Elements accessed by giving subscript for each dimension:

```cpp
matrix [i][j] = matrix [i-1][j+1] + 1;
```
Arrays and Pointers

Close relationship between arrays and pointers.

Array name is in fact treated just like a pointer.

Pointers can be subscripted, as if they were arrays (even if they aren’t!)

MonthNames + 3 is legal, means address of MonthNames[3].
Arrays as Arguments

When used as an argument, size need not be specified:

```c
int arraySum (int values[], int n)
    // compute sum of array values[0] .. values[n-1]
{
    int result = 0;
    for (int i = 0; i < n; i++) {
        result += values[i]
    }
    return result;
}
```
Structures

A structure is a collection of fields, which need not have the same type.

```cpp
struct person {
    string name;
    int age;
    enum {male, female} sex;
};
```

Fields are accessed using dot notation.

```cpp
person employee;
employee.name = "sam smith";
employee.age++;
if (employee.sex == male)
    ...
```

We actually won’t use structures, will use more general mechanism called `class` (Described in chapter 2).
Functions

Functions encapsulate a set of actions, so that later we can refer to
the sequence of actions by name alone:

```cpp
int Fahrenheit(int cTemp)
{
    return (cTemp * 9.0) / 5.0 + 32;
}
```

Parts:

- Header – with return type, name, and arguments
- Body – with statements to execute. Can have return statement
to end execution.

Return type can be void – no value.
A function prototype is a declaration but not a definition, just gives
name, arguments and return type.

```
// prototype for Fahrenheit – definition occurs later
int Fahrenheit (int);
```
Local Variables

Variables within a function come into existence when the function is entered, disappear when the function exits. Execute in stack-like fashion. Assume function A calls function B which calls function C – can imagine variables as follows:

<table>
<thead>
<tr>
<th>local variables for C</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables for B</td>
</tr>
<tr>
<td>local variables for A</td>
</tr>
</tbody>
</table>

Will eventually encounter recursive functions, functions that can call themselves. Imagine B is recursive, and has called itself once before calling C, can envision the following:

<table>
<thead>
<tr>
<th>local variables for C</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables for B</td>
</tr>
<tr>
<td>local variables for B</td>
</tr>
<tr>
<td>local variables for A</td>
</tr>
</tbody>
</table>
The Main Event

A program must always include a procedure named \texttt{main}, which is the starting point for execution.

\begin{verbatim}
#include <iostream>

void main() {
    // program to write table of squares
    cout << "Table of Squares\n";
    for (int i = 0; i < 12; i++) {
        cout << "i: " << i << " i squared " << \n         i * i << "\n";
    }
}
\end{verbatim}
Include Files

Many data structures require one to define an include file before they can be processed.

<table>
<thead>
<tr>
<th>purpose</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream input/output</td>
<td>iostream</td>
</tr>
<tr>
<td>math functions</td>
<td>math.h</td>
</tr>
<tr>
<td>complex numbers</td>
<td>complex</td>
</tr>
<tr>
<td>Boolean values</td>
<td>bool.h</td>
</tr>
<tr>
<td>generic algorithms</td>
<td>algorithm</td>
</tr>
</tbody>
</table>