

# **C++ Builder 4**

## **Client/Server Foundations**

**Borland**  
**INPRISE**

### **Courseware Manual**

### **Chapter 6**

Version 1.0  
Copyright © 1999  
Inprise Corporation  
All Rights Reserved

# *Chapter 6: The C++ Language*

---

## **What will be covered in this chapter:**

- ❑ How to create simple C++ Builder programs as exercises in C++
- ❑ The fundamental control elements in C++, including logical and mathematical operators, decision structures, and looping controls
- ❑ The data types that are available in C++, including numeric, string, Boolean, and user-defined types
- ❑ The facilities in C++ to create complex data structures, including arrays, sets, and structs

# Overview

---

This chapter entails the most basic fundamentals of the C++ language and serves as a brief overview of C++. Do not, however, consider this an exhaustive treatment of the language—that's what the language reference manual provides. Instead, this chapter aims to provide a basic understanding of the elements of C++ and how to work with them. If you are interested in pursuing some of the topics that are only briefly discussed or not reviewed at all, you can turn to the on-line language reference included as part of the C++ Builder Language Guide in the Help System.

# Language Basics

---

Before looking at the language and structures available in C++, there are some basic elements that broadly apply to all parts of C++.

## Comments

---

You have already seen some comments in earlier code examples, but we need to formalize the definition. There are three types of comment markers in C++:

Beginning Symbol	Ending Symbol	Scope
/*	*/	Everything between the braces is ignored
//		Everything from the comment marker to the end of the current line is ignored

You can use either set of comment markers but you cannot start a comment with one type and finish it with the other. C++ comments can span multiple lines (except, of course, the “//” marker). Also, you cannot embed close comment symbol `*/`, within a comment—it will always mark the end of the comment. In other words, you cannot nest comments.

## Statements

---

Programs and functions are composed of *statements* that are each an executable line of code. Statements can be simple or compound. Examples of simple statements are assignments and procedure or function calls.

```
X = 13; // assigns the value of 13 to variable X
Greeting = "Hi"; //assigns 'Hi' to Greeting
MyFunction(); //executes MyFunction
```

You can also have compound statements anywhere in which you have simple statements. A compound statement is a grouping of statements treated as a block. The start of a compound statement is always delimited by `{`, and the end of a compound statement is delimited by the reserved word `}`. Here is an example of a compound statement:

```
if (X == 13)
{
    ShowMessage( "X made it to Thirteen!" );
    X = 14;
}
```

Anywhere in a C++ program that a simple statement appears, a compound statement can appear as well. In the following examples, you will see the word statement written as:

```
<statement>;
```

This indicates that either a simple or compound statement can legally appear there.

## The Semi-colon

---

The notorious semi-colon gets its own section because programmers new to C++ seem to have the most problems with it. If you come from a XBase or Basic background, it seems unusual to place a semi-colon after every line of code. In XBase, the semi-colon is used to continue a line of code onto a subsequent line. Notice that in C++ you can freely create multiple lines of code that make up a single statement. To make matters worse, there are some situations where you cannot put a semi-colon.

So here is the rule: ***Only simple statements in a C++ program must be separated by a semi-colon.***

```
if (X == 13)
{
    ShowMessage( "X made it to Thirteen! " );
    X = 14; // semi-colon is required
}        // semi-colon is not required (this ends the
        //      compound statement)
```

# Fundamental Control Elements

---

This section covers the basic elements that make up any C++ program including the operators, decision, and looping constructs. Basic input and output are also mentioned. This section covers the major language elements of C++.

## Operators

---

C++ has most of the operators people expect from a modern language.

### *Assignment vs. Equality*

The assignment operator in C++ is the equal sign (“=”). It is distinct from the equal equal sign (“==”) which is used to test for equality and for type definitions.

### *Numeric*

The following table lists all of the numeric operators in C++.

Operator	Operation	Operand Types	Result Type
+	Addition	integer or floating point	integer or floating point
-	Subtraction	integer or floating point	integer or floating point
*	Multiplication	integer or floating point	integer or floating point
/	Division	integer or floating point	integer or floating point
%	Modulus (Integer remainder)	integer	integer

### *Relational*

The following table contains C++ Builder’s built-in relational operators. All of these operators return Boolean (**true** or **false**) values.

Operator	Operation
<	Less Than
>	Subtraction
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Here is a quick summary of the rest of the operators available in C++ Builder. For a more detailed review of operators available, consult the on-line help.

Bitwise

<<      Shift left

	>>	Shift right
	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
		Bitwise inclusive OR
Logical	&&	Logical AND
		Logical OR
Assignment	=	Assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	=	Assign bitwise OR
Class-member	::	Scope access/resolution
	.*	Dereference pointers to class member
	->*	Dereference pointers to pointers to class member
Conditional	?:	Actually a ternary operator for example, max = (x > y) ? x : y      "if x > y then max = x else max = y"
Comma	,	Separates elements in a function list argument

## Decision

---

C++ supports a number of decision structures, each one tailored to a particular use.

### *if...else*

The basic format of the **if...else** statement is shown below:

```
if (<logical expression>
    <statement>;
```

This basic syntax can be enhanced by adding an **else** clause:

```
if (<logical expression>) <statement>;
else
    <statement>;
```

Notice that in both of the above examples, <statement> can be a simple or compound statement.

Another variation on this statement is the possibility of additional **if** clauses following the **else**.

```
if (<logical expression>) <statement>;  
else if (<logical expression>) <statement>;
```

## *switch* <x> case

The **if...else** construct shown above works well for a small number of options, but becomes cumbersome if you have many more. In this situation, the **switch** construct is easier to write. The general syntax for the **switch** statement is shown below:

```
switch ( <switch variable> ) {  
  
    case <constant expression> : <statement>; [break;]  
    .  
    .  
    .  
    default : <statement>;  
  
}
```

The **Case** statement takes a variable and compares it with possible values. Multiple values can appear under the same **case**, separated by commas. Here is an example of a **Case** construct:

```
switch (ch)  
{  
    case 'a' :  
        printf("\nOption a was selected.\n");  
        break;  
    case 'b' :  
        printf("\nOption b was selected.\n");  
        break;  
    default :  
        printf("\nNOT A VALID CHOICE! Bye ...");  
        return(-1);  
}
```

In this example, we are checking which option a or b was selected. If neither was selected then we print not a valid choice.

Also note that any of the statements in a **switch** construct can be simple or compound statements. If you need to execute multiple lines of code for a given case, you must bracket that code in a { } pair.

## Looping

---

Looping is one of the fundamental activities that you will employ in a programming language, and C++ offers several alternatives, each particularly suited to a specific circumstance.

### *while*

The **while** loop executes statements until the looping condition becomes false. Here is the general syntax of the **while** loop:

```
while ( <condition> ) <statement>;
```

As in all structures, you can have simple or compound statements following the **<condition>**. Here is an example of a simple statement:

```
while (*p == ' ') p++;
```

This example will increment p until \*p no longer evaluates to ' '.

## *for*

The **for** loop is different from the **while** loop in that it always loops a specified number of time, determined by the control variable. The general syntax of the **for** loop is shown below:

```
for ( [<initialization>] ; [<condition>] ; [<increment>] )  
{ <statement>; }
```

Here, the control variable is usually an integer (or one of the integer types) and is frequently named “i” (which stands for index). This variable usually is initialized with a value of 1, but can start with any value. Therefore, to create a loop that runs from 0 to 100:

```
for (int i = 0; i < 100; i++)  
{  
    if (i == 8)  
        cout << "\ni = " << i;  
}
```

As this loop executes, the value of i is 0 the first time through the loop. Each time through the loop, i is incremented. The loop continues until i reaches the value of 100 (the final value here). The last time through the loop, i is equal to 100. It is incremented again, and the loop terminates because the value of i is now greater than the final value.

**for** loop construction allows for the increment expression to be any number of valid expressions. -- (minus minus) allows for the count down option for instance.

The **for** loop is best for looping when you know in advance how many times you will go through the loop. It is frequently used to access the elements of an array.

## *do...while*

The third of the looping constructs is the **do...while** loop whose general syntax is shown below:

```
do { <statement>; } while ( <condition> );
```

The **do...while** loop executes all of the statements within the loop until the condition evaluates to false. Notice that the **do...while** loop can contain simple or compound statements.

An example of a **do...while** loop:

```
do {
    printf ("Enter password: ");
    scanf ("%s", password);
} while (strcmp(password, checkword));
```

This loop will read in something typed until password = checkword.

The **do...while** loop is perfectly suited for situations where you want the code within the loop to always execute at least once. The condition that determines when the loop will terminate is not evaluated until the Loop statements have run once.

## *break and continue*

The last of the looping topics are the two reserved words **break** and **continue**. Either of these commands can be used in any of the above looping structures.

**Break** is used to immediately exit the loop that contains it. So, **break** is a convenient way to exit the loop without having to set special loop conditions. Here is an example:

```
switch (ch)
{
    case 'a' :
        printf ("\nOption a was selected.\n");
        break;
    case 'b' :
        printf ("\nOption b was selected.\n");
        break;
    default :
        printf ("\nNOT A VALID CHOICE! Bye ...");
        return(-1);
}
```

In this example, when a **break** is encountered, program execution jumps to the first statement following the loop.

The other loop execution modifier is the **Continue** statement. **Continue** will skip the rest of the loop and go back to the evaluation condition. So, if something changes that may affect the condition, you can use **continue** to jump to the top of the loop and evaluate the condition. Here is a simple example:

```
void main ()
{
    for (i = 0; i < 20; i++) {
        if (array[i] == 0)
            continue;
        array[i] = 1/array[i];
    }
}
```

The **Continue** statement in this case will prevent a divide by zero error.

**Continue** is particularly useful for skipping invalid input. If the user has entered something contrary to what you wanted, you can use **continue** to jump back to the top of the loop and get the input again.



# Data Types

---

In the following section, we are going to look at the standard built-in types and discuss how to create your own new types.

## Numeric

---

There are two distinct types of numeric types in C++: integral types and floating point types.

Ordinal numeric types are integers and there are a variety of different integer types available. Integers are always whole numbers (having no fractional part).

Numeric Types

Integral types

Type	Range	Internal Format
short	-32768 .. 32767	signed 16-bit
int, long		-2147483648 .. 2147483647 signed 32-bit

Floating Point Numbers

Type	Range	Size
float	$3.4 * (10^{**-38})$ to $3.4 * (10^{**+38})$	4 bytes
double	$5.0 * 10E-324$ .. $1.7 * 10E308$	8 bytes
long double	$3.410E-4932$ .. $1.1 * 10E4932$	80 bits

## Char

---

The char type in C++ is exactly what it sounds like—a holder for a single character. The char type can hold any ASCII character. Use the type specifier char to define a character data type. Variables of type char are 1 byte in length.

A char can be signed, unsigned, or unspecified. By default, signed char is assumed. Objects declared as characters (char) are large enough to store any member of the basic ASCII character set.

Type	Converts to	Method
char	int	Zero or sign-extended (default char type)
unsigned char	int	Zero-filled high byte (always)
signed char	int	Sign-extended (always)
short	int	Same value; sign extended
unsigned short	unsigned int	Same value; zero filled
enum	int	Same value

## Boolean

---

Boolean types are variables that can only take on the values **true** or **false**. **true** and **false** are both built-in identifier constants in C++, so they do not need to be defined, nor do they require any special notation.

## Pointers, User-defined types, and Sub-types

---

A complete discussion of these data types is beyond the scope of this class. However, we will mention them here. If you need more information about these data types, consult the manuals that come with C++ Builder.

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer classes have distinct properties, purposes, and rules for manipulation, although they do share certain C++Builder operations. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

C++ also allows you to create your own types using enum.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;

/* establishes a unique integral type, enum days, a variable
anyday of this type, and a set of enumerators (sun, mon,...) with
constant integer values. */

enum modes { LASTMODE = -1, BW40=0, C40, BW80, C80, MONO = 7 };

/*
"modes" is the type tag.
"LASTMODE", "BW40", "C40", etc. are the constant names.
The value of C40 is 1 (BW40 + 1); BW80 = 2 (C40 + 1), etc.
*/
```

## Variant

---

The Variant class is modeled after the intrinsic Variant type in previous versions of Delphi and Borland C++. Variant objects are essentially typeless variables – they can take on different types automatically. All other variables are restricted to their declared data types. However, the variant type allows a variable to change its type dynamically (at run-time). For example, in the following snippet of code, the variant V is first recognized as an integer, then as a string.

```
Variant V(OPENARRAY(int,(0,HighVal,0,HighVal)),varInteger);
```

A variant type variable can contain boolean, date/time, integer, OLE Automation objects, real, and string values. It can also store arrays with elements of these types, including variant arrays. There are two special values that can be used as variant values: *Null* and *Unassigned*. If the variant value is *NULL*, the variable

has unknown or missing data. The *unassigned* value means that the variable has not been assigned a value.

Variant objects are convenient but very inefficient. They are designed to hold references to OLE Automation server objects. The variant type variable can then be used to get and set properties of the OLE Automation object or to execute the object's methods.

For more information on the Variant class, consult the on-line help.

# Data Structures

---

We have seen the basic types that are available in C++. Now, let's see how to group simple types into aggregates or data structures. We will investigate arrays, structs, and classes as ways to group related data types together.

## Arrays

---

The simplest of these data structures is the array type. Arrays provide a way to create one variable that can contain multiple values, as in a list. Arrays are always typed in C++, meaning that all of the array elements must be of the same type. Here is the general syntax for declaring an array type:

```
type declarator [<constant-expression>]
```

Here are a few sample declarations:

```
int MyArray[10]; // This array holds 11 ints

long double **data;

try { // TEST FOR EXCEPTIONS.
    data = new long double*[m]; // STEP 1: SET UP THE
                                // ROWS.
    for (int j = 0; j < m; j++)
        data[j] = new long double[n]; // STEP 2: SET UP THE
                                        // COLUMNS
}
```

The last declaration creates a two dimensional array. You can also see how to reference the elements of the array. Also note that arrays in C++ are zero based, meaning the first element is at array location 0. To access the first integer in the MyArray integer array declared above you would access MyArray[0].

Arrays can contain any type of variable. However, in C++, an array can only hold one type of variable per array. Thus, you cannot have an array that holds integers and strings unless you create an array of variant type. Each array has a specific type of element that is allowed.

## Dynamic Arrays

Dynamic arrays are arrays that do not have a fixed size. Memory for a dynamic array is allocated as needed (when a value is assigned to an element of the array).

A Dynamic Array is a resizable collection implemented as an array. C++Builder provides the DynamicArray template as an implementation of Dynamic Arrays. The DynamicArray template uses a binary layout compatible with Delphi's Dynamic Array implementation, enabling you to share DynamicArrays of types compatible with both Delphi and C++Builder using Pascal and C++ code.

The syntax for declaring a dynamic array is as follows:

```
template <class T> class DELPHIRETURN DynamicArray;
```

Here is a sample declaration:

```
DynamicArray<int> arrayOfInt;  
  
arrayOfInt.Length = 10;  
cout << "ArrayLength: " << arrayOfInt.Length << endl;
```

To free a DynamicArray, assign it a Length of zero.

## Structs

---

The struct data type is a way to collect a logical group of values of differing types under the same name. For example, you can create a customer struct that includes the name, phone number, and height under a single identifier. Each of the data variables that are associated with the struct are known as *fields*.

Here is the basic syntax for creating records:

```
struct [<struct type name>] {  
  
    [<type> <variable-name[, variable-name, ...]>] ;  
    .  
    .  
    .  
  
} [<structure variables>] ;
```

You can have as many different fields of as many different types as you like. If you have multiple fields of the same type, you can separate them with commas and use the same type designator. Here is an example of the customer record discussed above:

```
#include <string.h>  
  
struct my_struct {  
    char name[80], phone_number[80];  
    int age, height;  
} my_friend;  
  
void func() {  
    strcpy(my_friend.name, "Mr. Wizard"); /* accessing an element  
*/  
}  
  
struct my_struct my_friends[100]; /* declaring additional  
variables */
```

To access elements in a structure, use a record selector (.) or ->.

To declare additional variables of the same type, use the keyword struct followed by the <struct type name>, followed by the variable names. In C++ the keyword struct can be omitted.



# Summary

---

## **What was covered in this chapter:**

- ✓ We saw how to create simple C++ Builder programs to exercise C++ in C++ Builder without having to create forms.
- ✓ We looked at the fundamental control elements in C++, including the operators, decision structures, and looping controls.
- ✓ We also looked at the basic data types that are available in C++, including numeric, Boolean, and user-defined types.
- ✓ We discovered the facilities in C++ to create complex data structures, including arrays, and structs.

## CBUILDER 4 FOUNDATION COURSEWARE NO-NONSENSE LICENSE STATEMENT AND DISCLAIMER

### IMPORTANT – READ CAREFULLY

This license statement and disclaimer constitutes a legal agreement (“License Agreement”) between you (an individual) and Inprise Corporation (“Inprise”) for the Cbuilder 4 Foundation Course (“Courseware”), including any software, media, and accompanying on-line or printed documentation. This License Agreement and your rights are only for each of the first twelve chapters of the Courseware that are being provided at no charge under this special program.

**BY INSTALLING, COPYING, OR OTHERWISE USING THE COURSEWARE, YOU AGREE TO BE BOUND BY ALL OF THE TERMS AND CONDITIONS OF THE LICENSE AGREEMENT.**

Upon your acceptance of the terms and conditions of the License Agreement, Inprise grants you the right to use the Courseware in the manner provided below.

This Courseware is owned by Inprise or its suppliers and is protected by copyright law and international copyright treaty. Therefore, you must treat this Courseware like any other copyrighted material (e.g., a book), except that you may either make one copy of the Courseware solely for backup or archival purposes or transfer the Courseware to a single hard disk provided you keep the original solely for backup or archival purposes.

You may transfer the Courseware and documentation on a permanent basis provided you retain no copies and the recipient agrees to the terms of the License Agreement. Except as provided in the License Agreement, you may not transfer, rent, lease, lend, copy, modify, translate, sublicense, time-share or electronically transmit or receive the Courseware, media or documentation.

### WARRANTY DISCLAIMER

**THE COURSEWARE ARE PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND.**

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, BORLAND AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT, WITH REGARD TO THE COURSEWARE. SOME STATES/JURISDICTIONS DO NOT ALLOW LIMITATIONS ON DURATION OF AN IMPLIED WARRANTY, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

### LIMITATION OF LIABILITY

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL BORLAND OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE COURSEWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF BORLAND HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, BORLAND'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS LICENSE AGREEMENT SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE COURSEWARE PRODUCT OR U.S. \$25. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

### U.S. GOVERNMENT RESTRICTED RIGHTS

The Courseware and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraphs (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Inprise Corporation., 100 Enterprise Way, Scotts Valley, CA 95066.

---

#### GENERAL PROVISIONS

This statement may only be modified in writing signed by you and an authorized officer of Inprise. If any provision of this statement is found void or unenforceable, the remainder will remain valid and enforceable according to its terms. If any remedy provided is determined to have failed for its essential purpose, all limitations of liability and exclusions shall remain in effect.

This statement shall be construed, interpreted and governed by the laws of the State of California, U.S.A. This statement gives you specific legal rights; you may have others which vary from state to state and from country to country. Inprise reserves all rights not specifically granted in this statement.

Form 05/13/00

---