How to: Use Basic C++, Syntax and Operators

A

In this How to we summarize the basic syntax of C++ and the rules and operators that are used in C++ expressions. This is a brief language reference for C++ as used in this book, not the entire language. We don't cover the use of dynamic memory in this How to.

A.1 Syntax

A.1.1 The Function `main`

C++ programs begin execution with a function `main`. The smallest C++ program is shown below. It doesn't do anything, but it's syntactically legal C++.

```
int main()
{
    return 0;
}
```

The return value is passed back to the "system," a nonzero value indicates some kind of failure. Although not used in this book, the function `main` can have parameters, these are so-called **command-line parameters**, passed to the program when the function is run. A brief synopsis of command-line parameters is given in Section A.2.6.

A.1.2 Built-in and Other Types

The built-in types in C++ used in this book are `int`, `double`, `bool`, and `char`. A type `void` is used as a built-in "empty-type," but we don't treat `void` as an explicit type. A function declared as `void` does not return a value of type `void`, but returns no value. In addition to `double`, the type `float` can be used to represent floating point values; but `float` is less precise and we do not use it. We also don't use the modifiers `short`, `long`, and `unsigned`. The most useful of these is `unsigned` which effectively doubles the range of positive values, (e.g., on many systems a `char` value is signed and ranges from −128 to 128, but an `unsigned char` value ranges from 0 to 255).

The range of values represented by `int` and `char` expressions are accessible using values given in the header file `<climits>` (or `<limits.h>`). The range for floating point values is found in `<cfloat>` (or `<float.h>`.) See How to F for information about these standard header files.

Types other than built-in types in C++ are called programmer-defined types. Many programmer-defined types are implemented for use in this book. Information about these types can be found in How to G. Standard C++ user-defined types used in this book include `string` and `vector`. We use a class `tvector` with error-checking rather than the standard vector class, but we use the standard C++ string class declared in `<string>`. For programmers without access to this class we provide a class `tstring` that can be used in place of `string` for all the programs in this book. The class `tstring` is accessible via the header file `"tstring.h"`.

A.1.3 Variable Definition and Assignment

Variables are **defined** when storage is allocated. Variable definitions include a type and a name for the variable. An initial value may optionally be supplied. The C++ statements below define an `int` variable without an initial value, a `double` variable with an initial value, two string variables with initial values and one without.

```
int minimum;
double xcoord = 0.0;
string first = "hello", second, third="goodbye";
```

In this book we usually define only one variable per statement, but as the string definitions above show, any number of variables can be defined for one type in a definition statement. One good reason to define only one variable in a statement is to avoid problems with pointer variables. The statement below makes `foop` a pointer to a `Foo`, but `foop2` is a `Foo`, not a pointer.

```
Foo * foop, foop2;
```

To make both variables pointers requires the following.

```
Foo * foop, * foop2;
```

Variables that are instances of a class, as opposed to built-in types like `int` or `bool`, are constructed when they are defined. Typically the syntax used for construction looks like a function call, but the assignment operator can be used when variables are defined as in the second line below. This statement constructs a variable only, it does not construct then assign, although the syntax looks like this is what happens.

```
Dice cube(6);       // construct a 6-sided Dice
Dice dodo = 12;     // construct a 12-sided Dice
Date usb(1,1,2000); // construct Jan 1, 2000
```

It's legal to use constructor syntax for built-in types too:

```
 int x(0);
 int y = 0;  // both define ints with value zero
```

Table A.1 Arithmetic Assignment Operators.

Symbol	Example	Equivalent
`+=`	`x += 1;`	`x = x + 1;`
`*=`	`doub *= 2;`	`doub = doub * 2;`
`-=`	`n -= 5;`	`n = n - 5;`
`/=`	`third /= 3;`	`third = third / 3;`
`%=`	`odd %= 2;`	`odd = odd % 2;`

The Assignment Operator. The assignment operator, `operator =`, assigns new values to variables that have *already been defined*. The assignment operator assigns values to `tomorrow` and `baker` below.

```
Date today;
Date tomorrow = today + 1;  // definition, not assignment
int  dozen = 12, baker = 0; // definition, not assignment
tomorrow = today - 1;    // make tomorrow yesterday
baker    = dozen + 1;    // a triskaidekaphobe’s nightmare
```

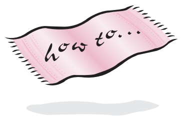
Assignments can be chained together. The first statement using `operator =` below shows a single assignment, the second shows chained assignment.

```
double average;
int min,max;

average = 0.0;
min = max = ReadFirstValue();
```

The assignment of 0.0 to `average` could have been done when `average` is defined, but the assignments to `min` and `max` cannot be done when the variables are defined since, presumably, the function `ReadFirstValue` is to be called only once to read a first value which will then be assigned to be both `min` and `max`.

In addition to the assignment operator, several arithmetic assignment operators alter the value of existing variables using arithmetic operations as shown in Table A.1.



The expectation in C++ is that an assignment results in a copy. For classes that contain pointers as data members, this usually requires implementing/overloading an assignment operator and a copy constructor. You don’t need to worry about these unless you’re using pointers or classes that use pointers. See How to E for details on overloading the assignment operator.

Table A.2 C++ keywords

asm	default	for	private	struct	unsigned
auto	delete	friend	protected	switch	using
bool	do	goto	public	template	virtual
break	double	if	register	this	void
case	dynamic_cast	inline	reinterpret_cast	throw	volatile
catch	else	int	return	true	wchar_t
char	enum	long	short	try	while
class	explicit	mutable	signed	typedef	
const	extern	namespace	sizeof	typeid	
const_cast	false	new	static	typename	
continue	float	operator	static_cast	union	

A.1.4 C++ Keywords

The keywords (or reserved words) in C++ are given in Table A.2. Not all the keywords are used in this book. We either discuss or use in code all keywords except for the following:

asm, auto, goto, register, throw, volatile, catch, wchar_t, short, try, extern, typeid, typename, union

A.1.5 Control Flow

We use most of the C++ statements that change execution flow in a program, but not all of them. We use the statements listed in Table A.3.

Table A.3 Control flow statements in C++

if (*condition* **)** *statement*
if (*condition* **)** *statement* **else** *statement*
switch (*condition* **)** *case/default statements*

while (*condition* **)** *statement*
do *statement* **while (** *condition* **)**
for (*init statement* ; *condition* ; *update expression* **)**

case *constant expression* **:** *statement*
default : *statement*
break;
continue;
return *expression* (expression is optional)

There are few control statements we do not use, these are:

- **try** and **catch** for handling exceptions. We do not use exceptions in the code in this book, so we do not need try and catch.
- **goto** for jumping to a labelled statement. Although controversial, there are occasions where using a goto is very useful. However, we do not encounter any of these occasions in the code used in this book.

Selection Statements. The `if` statement by itself guards a block of statements so that they're executed only when a condition is true.

```
if (a < b)
{  // statements only executed when a < b
}
```

The `if/else` statement selects one of two blocks of statements to execute:

```
if (a < b)
{ // executed when a < b
}
else
{ // executed when a >= b
}
```

It's possible to chain `if/else` statements together to select between multiple conditions, (see Section 4.4.2). Alternatively, the `switch` statement selects between many *constant* values, the values must be integer/ordinal values, (e.g., ints, chars, and enums can be used, but doubles and strings cannot).

```
switch (expression)
{
  case 1 :
     // do this
     break;
  case 2 :
     // do that
     return;
  case 20:
  case 30:
     // do the other
     break;
  default:
     // if no case selected
}
```

Conditional Expressions: the `?:` operator. Although not a control statement, the question-mark/colon operator used in a **conditional expression** replaces an `if/else` statement that distinguishes one of two values. For example, consider the statement below.

```
  if (a < b)    // assign to min the smallest of a and b
  {   min = a;
  }
  else
  {   min = b;
  }
```

This can be expressed more tersely by using a conditional:

```
// assign to min the smallest of a and b
  min = (a < b) ? a : b;
```

A conditional expression consists of three parts, a condition whose truth determines which of two values are selected, and two expressions for the values. When evaluated, the conditional takes on one of the two values. The expression that comes before the question-mark is interpreted as boolean-valued. If it is true (non-zero), then *expression a* is used as the value of the conditional, otherwise *expression b* is used as the value of the conditional.

Syntax: conditional statement (the ?: operator)

condition expression ? *expression a* : *expression b*

We do not use `operator ?:` in the code shown in the book although it is used in some of the libraries provided with the book, (e.g., it is used in the implementation of the `tvector` class accessible in `tvector.h`).

Repetition. There are three loop constructs in C++, they're shown in Table A.3. Each looping construct repeatedly executes a block of statements while a guard or test-expression is true. The while loop may never execute, (e.g., when `a > b` before the first loop test in the following).

```
  while (a < b)
  {  // do this while a < b
  }
```

A do-while loop always executes at least once.

```
  do
  {  // do that while a < b
  } while (a < b);
```

A for statement combines loop initialization, test, and update in one place. It's convenient to use for loops for definite loops, but none of the loop statements generates code that's more efficient than the others.

```
  int k;
  for(k = 0; k < 20; k++)
  { // do that 20 times
  }
```

It's possible to write infinite loops. The `break` statement branches to the first statement after the innermost loop in which the `break` occurs.

```
while (true)         for(;;)            while (true)
{ // do forever      { // do forever    {  if (a < b) break;
}                    }                  }
```

Occasionally the `continue` statement is useful to jump immediately back to the loop test.

```
while (expression)
{  if (something)
   { // do that
     continue;  // test expression
   }
   // when something isn't true
}
```

Function Returns. The `return` statement causes control to return immediately from a function to the statement following the function call. Functions can have multiple return statements. A `void` function cannot return a value, but `return` can be used to leave the function other than by falling through to the end of the function.

```
void dothis()                int getvalue()
{  if (test) return;         {  if (test) return 3;
   // do this                   // do that

}  // function returns      } // error, no value returned
```

A.2 Functions and Classes

A.2.1 Defining and Declaring Functions and Classes

A function is *declared* when its prototype is given and *defined* when the body of the function is written. A function's header must appear before the function is called, either as part of a function definition or as a prototype. Two prototypes follow.

```
int doThat(int x, int y);     void readIt(const string& s);
```

The return type of a function is part of the prototype, but isn't used to distinguish one function from another when the function is overloaded. Overloaded functions have the same names, but different parameter lists:

```
void check(int x);            void check(string s);
void check(bool a, bool b);   int check(int x);  // conflict
```

A class is *declared* when member function prototypes and instance variables are provided, typically in a header file. The body of the member functions aren't typically included as part of a declaration.

```
class Bug;     // forward declaration
class Doodle   // Doodle declaration only, not definition
{
  public:
    Doodle();
    Doodle(int x);
    int   getDoo() const;
    void  setDoo(int x) ;
  private:
    int   myDoo;
    Bug * myBug;
};
```

Functions that don't alter class state should be declared as `const`. See How to D for details. The class *definition* occurs in an implementation file, typically with a .cpp suffix.

```
int Doodle::getDoo() const   // method definition
{
   return myDoo;
}
```

It's possible to define member functions *inline* as part of the class declaration:

```
class Simple  // declaration and inline definitions
{
  public:
     Simple(const string& s) : myString(s) { }
     void Set(const string& s)
     {  myString = s;
     }
     int Get() const { return myString; }
  private:
     string myString;
};
```

The class `Simple` shows an initializer list used to construct private instance variables. Initializer lists are the preferred form of giving values to instance variables/data members when an object is constructed.

Initializer Lists. A class can have more than one constructor; each constructor should give initial values to all private data members. All data members will be constructed before the body of the constructor executes. Initializer lists permit parameters to be passed to data member constructors. Data members are initialized in the order in which they appear in the class declaration, so the initializer list should use the same order.

In C++ it's not possible for one constructor to call another of the same class. When there's code in common to several constructors it should be factored into a private `Init` function that's called from the constructors. However, each constructor must have its own initializer list.

A.2.2 Importing Classes and Functions: `#include`

Class and function libraries are typically compiled separately and linked with client code to create an executable program. The client code must import the class and function declarations so the compiler can determine if classes and functions are used correctly. Declarations are typically imported using the preprocessor directive `#include` which literally copies the specified file into the program being compiled.

The C++ standard specifies that standard include files are specified without a .h suffix, that is, `<iostream>` and `<string>`. For the most part, these header files import declarations that are in the *std* **namespace** (see Section A.2.3). Using a file with the .h suffix, for example `<iostream.h>`, imports the file in the global namespace. This means that the directives below on the left are the same as that on the right.

```
#include <iostream>                #include <iostream.h>
using namespace std;
```

Although most systems support both `<iostream>` and `<iostream.h>`, the namespace-version is what's called for in the C++ standard. In addition, some files do not have equivalents with a .h suffix—the primary example is `<string>`.

A.2.3 Namespaces

Large programs may use classes and functions created by hundreds of software developers. In large programs it is likely that two classes or functions with the same name will be created, causing a conflict since names must be unique within a program. The **namespace** mechanism permits functions and classes that are logically related to be grouped together. Just as member functions are specified by qualifying the function name with the class name, as in `Dice::Roll` or `string::substr`, functions and classes that are part of a namespace must specify the namespace. Examples are shown in Program A.1 for a user-defined namespace *Math*, and the standard namespace *std*. Note that `using namespace std` is not part of the program.

Program A.1 namespacedemo.cpp

```
#include <iostream>

// illustrates using namespaces

namespace Math
{
  int factorial(int n);
  int fibonacci(int n);
}

int Math::factorial(int n)
// post: return n!
{
    int product = 1;
    // invariant: product = (k-1)!
    for(int k=1; k <= n; k++)
    {    product *= k;
    }
    return product;
}

int Math::fibonacci(int n)
// post: return n-th Fibonacci number
{
    int f = 1;
    int f2= 1;
    // invariant: f = F_(k-1)
    for(int k=1; k <= n; k++)
    {   int newf = f + f2;
        f = f2;
        f2 = newf;
    }
    return f;
}

int main()
{
   int n;
   std::cout << "enter n ";
   std::cin >> n;

   std::cout << n << "! = "         << Math::factorial(n) << std::endl;
   std::cout << "F_(" << n << ")= " << Math::fibonacci(n) << std::endl;

   return 0;
}
```

namespacedemo.cpp

OUTPUT

```
prompt> namespacedemo
enter n 12
12 = 479001600!
F_(12)= 233
```

Writing `std::cout` and `std::endl` each time these stream names are used would be cumbersome. The **using** declaration permits all function and class names that are part of a namespace to be used without specifying the namespace. Hence all the programs in this book begin with `using namespace std;` which means function and class names in the standard namespace *std* do not need to be explicitly qualified with `std::`. When you write functions or classes that are not part of a namespace, they're said to be in the **global namespace**.

A.2.4 Operators

Table A.4 C++ Operator Precedence and Associativity

operator symbol	name/description	associativity
::	scope resolution	left
()	function call	left
[]	subsccript/index	left
.	member selection	left
->	member selection (indirect)	left
++ –	post increment/decrement	right
dynamic_cast<type>		right
static_cast<type>		right
const_cast<type>		right
sizeof	size of type/object	right
++ –	pre increment	right
new	create/allocate	right
delete	destroy/de-allocate	right
!	logical not	right
- +	unary minus/plus	right
&	address of	right
*	dereference	right
(type)	cast	right
.* ->*	member selection	left
* / %	multiply, divide, modulus	left
+	plus (binary addition)	left
-	minus (binary subtraction)	left
<<	shift-left/stream insert	left
>>	shift-right/stream extract	left
< <= > >=	relational comparisons	left
== !=	equal, not equal	left
&	bitwise and	left
ˆ	bitwise exclusive or	left
|	bitwise or	left
&&	logical and	left
||	logical or	left
=	assignment	right
+= -= *= /= %=	arithmetic assignment	right
<<= >>=	shift assign	right
? :	conditional	right
throw	throw exception	right
,	sequencing	left

The many operators in C++ all appear in [Str97] Table A.4. An operator's precedence determines the order in which it is evaluated in a complex statement that doesn't use parentheses. An operator's associativity determines whether a sequence of connected operators is evaluated left-to-right or right-to-left. The lines in the table separate operators of the same precedence.

A.2.5 Characters

Characters in C++ typically use an ASCII encoding, but it's possible that some implementations use UNICODE or another encoding. Table F.3 in How to F provides ASCII values for all characters. Regardless of the underlying character set, the escape sequences in Table A.5 are part of C++.

The newline character `\n` and the carriage return character `\r` are used to indicate end-of-line in text in a platform-specific way. In Unix systems, text files have a single end-of-line character, `\n`. In Windows environments two characters are used, `\n\r`. This can cause problems transferring text files from one operating system to another.

A.2.6 Command-line Parameters

Command-line parameters are not covered in this book, but Program A.2, *mainargs.cpp* shows how command-line parameters are processed by printing each parameter. Parameters are passed in an array of C-style strings conventionally named `argv` (argument vector). The number of strings is passed in an int parameter named `argc` (argument count). Every program has one parameter, the name of the program which is stored in `argv[0]`.

Table A.5 Escape sequences in C++

escape sequence	name	ASCII
`\n`	newline	NL (LF)
`\t`	horizontal tab	HT
`\v`	vertical tab	VT
`\b`	backspace	BS
`\r`	carriage return	CR
`\f`	form feed	FF
`\a`	alert (bell)	BEL
`\\`	backslash	\
`\?`	question mark	?
`\’`	single quote (apostrophe)	’
`\"`	double quote	"

Program A.2 mainargs.cpp

```
#include <iostream>
using namespace std;

int main(int argc, char * argv[])
{
    int k;
    cout << "program name = " << argv[0] << endl;
    cout << "# arguments passed = " << argc << endl;

    for(k=1; k < argc; k++)
    {    cout << k << "\t" << argv[k] << endl;
    }
    return 0;
}
```

mainargs.cpp

OUTPUT

```
prompt> mainargs
program name = c:\book\ed2\code\generic\tapestryapp.exe
# arguments passed = 1
```

the run below is made from a command-line prompt, e.g., a Unix prompt, a DOS shell, or a Windows shell

```
prompt> ./tapestryapp this is a test
program name = ./tapestryapp
1    this
2    is
3    a
4    test
```

As you can see if you look carefully at the output, the name of the program is actually `tapestryapp`, although we've used the convention of using the filename of the program, in this case `mainargs`, when illustrating output.