An Overview of C and C++ Syntax

Terry Sergeant*

04 Aug 1998

*e-mail: sergeantt@alpha.obu.edu, web: www-sergeant.obu.edu

Contents

Preface

This manual provides a catalog of C language constructs and syntax along with short examples. It is intended to be a quick reference booklet. The reader who requires a more complete reference or tutorial is encouraged to obtain one or more books on the topic. An annotated bibliography can be found on page 34. Many of these books are likely available at local book stores, but would also almost certainly be available online at www.amazon.com.

It is assumed that the user of this manual has access to a C compiler and understands the process of compiling and running programs. This manual provides only minimal coverage of I/O commands and provides no information on building user-interfaces.

The manual is organized into two main parts. In part I the commands and constructs of C are introduced. Statements have been organized according to the three classical control control structures: sequence, selection, and repetition. Basically, anything that doesn't fall under the categories of selection or repetition, is considered to be "sequence". In part II some of the C++ extensions to C are introduced. C++ classes are the main feature of that section. This manual does not attempt to explain concepts of object-oriented programming beyond the what is necessary for understanding the syntax of using classes in C++. That is, classes are presented as "glorified" `struct`s and not as a paradigm shift from structured programming to object-oriented programming.

Part I
Plain Ol' C

This part of the manual is organized around the three classical control structures: sequence, selection, and repetition. That is, statements in a program happen in the order in which they appear (i.e., sequentially), unless they are altered by selection or repetition statements. Based on this characterization of statements, anything that does not fall under selection or repetition is regarded as sequence. Two additional sections have been added to keep the sequence section from becoming too cumbersome: Data Structures and Subprograms.

1 Introduction

The C programming language is designed to be a "small," fast language that provides the programmer with flexibility and power. Of course, with flexibility and power comes the ability to shoot oneself in the proverbial foot.

One of the nice features about C is that there are many built-in facilities for manipulating strings, performing complex calculations, and handling the system clock. To be able to use many of these functionalities, you are required to "include" a header file where these are defined. Some common header files are given below:

math.h	provides mathematical functions such as `power` and `ln`
stdio.h	basic I/O primitives such as `scanf` and `printf`
stdlib.h	some useful and powerful tools such as `qsort` and `rand`
time.h	ability to obtain and manipulate clock values (`clock`, `time`, etc.)
string.h	string manipulation functions such as `strcpy` and `strcat`

These header files are included in a program by putting the `#include` compiler directive at the beginning of the source code. **WARNING:** The `#` must be in the left-most column for it to be recognized as a compiler directive.

Example

This example shows how to include some common header files. Also notice the declaration of the main() function which where execution of a C program begins.

```
#include<stdio.h>
#include<string.h>
#include<math.h>

void main()
{
  .
  .
  .
}
```

Note

Most commands in C are terminated with a semi-colon. The `#include` directives are NOT terminated with a semi-colon because they are "compiler directives" instead of "statements".

Another Note

Identifiers in C are case-sensitive! That is, a variable named `score` is different from a variable named `Score` which is different that `SCORE` ... different than `ScOrE` ...

2 Sequence

2.1 Comments

In C, a comment section (to be ignored by the compiler) is started with `/*` and ended with `*/`. Comments created in this fashion can span multiple lines. Nesting comments may or may not be allowed depending on the compiler.

Example

This example shows both single- and multiple-line comments.

```
/*--------------------------------------------------
   Programmer   : Cher
   Last Modified: 28 Jul 1998
   Description  : This program ...

--------------------------------------------------*/
```

```
.
.
.
pi_are_square= pi * r * r;   /* area of a round pie */
```

2.2 Simple Data Types

Simple variables in C are declared to be of one of the following types:

Type	**Name**	**Memory**	**Approximate Range**
int	integer	32 bits	-2.14 billion — +2.14 billion
float	real	32 bits	1.2E-38 — 3.4E+38
double	real	64 bits	2.2E-308 — 1.8E+308
char	char	8 bits	256 different characters

The `float` and `double` data types can represent negative numbers as well.

Note

The mathematical functions that are found in `math.h` assume parameters will be of type `double`. Although variables of type `float` will be type cast automatically, it is conventional to use `double` unless there are severe memory limitations.

Another Note

Standard C handles boolean values as integers where a value of 0 is designated as false and non-zero values are true. C++ includes a `bool` type and constants `false` and `true`, but the implementation is with integers in the plain C fashion.

Example

To declare variables, list the type name followed by a list of identifiers. Notice that you can initialize variables at the declaration as with `p` and `q`. The other variables are uninitialized (i.e., who knows what they have in them!).

```
void main()
{
  int    a,b,c;    /* three integer variables       */
  double x,y,z;    /* three real-numbered variables */
  char   r;        /* one character variable        */
  int    p= 0, q= 12;  /* mind your p's and q's     */
  .
```

```
    .
    .
}
```

Warning 2.2

Floating point arithmetic (in any language) is only an approximation. For this reason, when dealing with monetary values it is a common practice to represent amounts in cents (i.e., integers) during calculations. The order of calculations can cause discrepancies in the values. The following code segment produced differing values for `x` and `y` even though the only difference is the order in which the calculations were performed.

```
#include<stdio.h>     /* necessary for printf */
void main()
{
  int i;
  float x,y;

  x= 0.0;
  y= 0.0;
  for (i=1; i<=1000; i++)
    x+= 100.0 / (float) (i * i);

  for (i=1000; i>=1; i--)
    y+= 100.0 / (float) (i * i);

  printf("%15.12f and %15.12f are ",x,y);
  if (x != y)
    printf("NOT ");
  printf("equal!\n");
}
```

The output of this program was:

```
164.393493652344 and 164.393463134766 are NOT equal!
```

2.3 Basic I/O

This section introduces the `printf` and `scanf` functions. These functions have numerous options which are not covered here. The purpose of this section is to provide enough I/O for debugging programs. To produce output to `stdout` (typically the monitor), you will use:

`printf("`*format-string*`",` *variable-list*`).`

The format string contains the actual characters you want to display and place-holders for any values obtained from variables or calculations. To specify a place-holder, use `%` followed by a type specifier. Some common specifiers are given below:

`%d`	integer (decimal)
`%f`	`float` or `double`
`%c`	character
`%s`	string

Example

Here are some standard examples of using `printf`. The output is *not* terminated with a newline character, so it is necessary to explicitly place that character into the format string. This is done with backslash followed by `n`. Smart alecks inevitably want to know how to show the percent sign since it is used to specify a place holder; that is handled by placing two consecutive %'s in the format string as demonstrated below.

```
#include<stdio.h>
int a=7;
double z= 8.9;

printf("I am %d years old\n",a);
printf("I am %f%% sure you are NOT %d years old\n",z,a);
```

The example found in warning 2.2 demonstrates how to specify the number of decimal places (12 in that example) when displaying a floating point value.

Input from `stdin` (typically the keyboard) is accomplish with the `scanf` statement as follows:

`scanf("`*format-string*`",` *variable-address-list*`).`

The format string uses the same %-codes as the `printf` function. Often, a format string contains only a single code. This is followed by the address(es) of the variable(s) into which the typed value should be placed. For more information about addresses see section 5.3.

Another Example

Here are some standard examples of using `scanf`. This example also demonstrates the usage of an array of `char`'s to create a "string". The variable name represents the address of the string, so the & is not necessary.

```
#include<stdio.h>
int    age;
double height;
char   name[80];

printf("Enter your age: ");
scanf("%d",&age);
printf("Enter your height (in feet): ");
scanf("%f",&height);
printf("Enter your name: ");
scanf("%s",name);
```

2.4 Assignment / Arithmetic

If computers can do anything, they can calculate. Although there are a number of powerful mathematical functions in `math.h`, this section focuses on the basic arithmetic operators. The operators are given below:

*	multiplication
/	division
%	modular division
+	addition
−	subtraction

Parentheses are used to override the default order of operations which follow the convention often taught and sometimes learned in algebra classes everywhere: multiplication and division first, then addition and subtraction, unless overridden by parentheses; ties are resolved from left to right.

The only "tricky" part has to do with variable types. When using `/` with integers, the result is always a truncated integer. The `%` operator always requires integer operands and can be used in conjunction with `/` to obtain a remainder along with its divisor (respectively). That is, `25 / 4` is `6` and `25 % 4` is `1`.

If you want to mix integers and real numbers in a single expression you will need to type cast the integers to floating point values.

Example

Type casting is performed by specifying the type to be changed in parenthesis before the variable. In this example, `b` is type cast to be a `double` before it is used in the calculation. Notice that the assignment operator is a single `=`. This is in contrast to the `==` operator which is used when comparing two values as described in section 2.5.

```
int    a=1,b=2,c;
double x=1.1,y=2.2,z;

a= a + 5*b;               /* a= 11  */
z= (double) b * x + y;    /* z= 3.3 */
```

Note

Some statements appear so often in programs that C has a shorthand notation for them. One example is the incrementing of a variable: `i= i + 1`. In C this can be written `i++`. The table below lists some common abbreviations. Keep in mind that there are equivalent operations for all of the arithmetic operators.

Shorthand	**Equivalent**
`i++`	`i= i + 1`
`i--`	`i= i - 1`
`i+= 5`	`i= i + 5`
`i/= x`	`i= i / x`

Another Note

The statement `i++` can also be rendered `++i` producing the same result. The former is the post-increment operator and the latter is the pre-increment operator. The difference in behavior only matter when the increment appears as part of a more complicated expression.

2.5 Comparison

Comparing values is necessary for selection and repetition constructs. The basic comparison operators are as follows:

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Examples of these operators are given in the discussion of `if` statements found in section 3.1.

3 Selection

Selection statements are used to perform (or omit) parts of the program based on the values of variables at run-time. Two methods for performing selection in C are `if` statements and `switch` statements.

3.1 `if` statements

An `if` statement is perhaps best introduced with examples:

Example

Notice that the condition being checked is placed within parentheses. The following statement will be performed if that condition evaluates to a non-zero value (and will be skipped if if the condition evaluates to a zero value). To perform multiple statements, the squiggley brackets ({}) are used to group the statements to be performed. The `else` keyword is optional.

```
#include<stdio.h>
int    a= 2,b= 5;

if (a == 23)
  printf("They're equal!\n");
else
  printf("They're NOT equal!\n");
```

```
if (a <= b)
{
  a*= b;
  printf("%d\n",a);  /* what gets printed here? */
}

if (a >= 100 && a <= 0) /* && means "and" ... why is this */
  printf("Ha!\n");      /* probably a logical error? */
else
  printf("Ha!\n");

if (!a)              /* what gets output here? */
  printf("NOT a\n");
```

The `||` and `&&` operators ("logical or" and "logical and" respectively) can be used to combine other boolean expressions into a more complex condition. The `!` operator is the "logical negation" operator and precedes the expression to be not-ed.

Warning

The following statement is syntactically correct, but is often a logical error. Can you figure out why?

```
int a= 0;
if (a = 7)
  printf("A is equal to 7\n");
```

3.2 `switch` statements

The switch statement is useful for selecting a choice based on comparison with a constant value. Consider the following example:

Example

In this example, the user is asked to enter a TV channel number. A response is generated based on their choice. If the entered value is equal to 1 then "ABC" gets printed. If the value is 5, 6, or 7 then "Sci-Fi" gets printed. If 4 is selected then "FOXSci-Fi" gets printed. So what is the purpose of the `break` statement? The `default` option is, well, the default.

```
int channel;
printf("Enter your favorite channel (integer): ");
scanf("%d",&channel);

switch(channel) {
  case 1 : printf("ABC"); break;
  case 2 : printf("NBC"); break;
  case 3 : printf("CBS"); break;
  case 4 : printf("FOX");  /* probably a mistake to */
  case 5 :                 /* leave off the break   */
  case 6 :                 /* statement here        */
  case 7 : printf("Sci-Fi"); break;
  default: printf("What?");
}
```

4 Repetition

Before loops can make any sense, selection statements (especially `ifs`) need to be clearly understood. The basic idea is this: every loop needs to have some condition, which when met, causes the loop to stop. We will consider three kinds of loops.

4.1 `for` loops

The basic form of a `for` loop in C is as follows:

`for (`*initialization*; *condition*; *updates*`)` *loop-body*

where

initialization	the statement(s) to be performed upon entry into the loop
condition	the condition to be checked each time through the loop; if the condition is true, the body of the loop is performed again
updates	the value(s) to be updated each time through the loop
loop-body	the statements to be performed each time through the loop

Example

The initialization and updates can contain several statements (separated by commas). The condition can be simple or complex. Watch out for misplaced semicolons as in the last example.

The middle loop contains a few subtleties that are worth mentioning. First notice, that `j` is not declared. The keyword `int` in the loop initialization allows a previously undeclared variable to be introduced (works in C++ only). Also notice that the loop condition contains a reference to `user` before it gets initialized. In this case that is okay because of "short-circuit" evaluation of boolean expressions. Therefore, the variable `user` doesn't actually get referenced until after the user has entered a value for the first time.

```
#include<stdio.h>
int i,user;

for (i=0; i<10; i++)
  printf("%d\n",i);

for (int j=0; j<10 || user; j++)
{
  printf("%d\n",j*2);
  if (!(j < 9))
  {
    printf("Enter 0 to quit, 1 to continue: ");
    scanf("%d",&user);
  }
}

user= 0;
for (i=25; i>0; i/=5,user++) ; /* what does this do?      */
  printf("User: %d\n",user);   /* i.e., what gets output? */
```

4.2 `while` loops

The basic form of `while` loops is as follows:

`while (`*condition*`)` *loop-body*

If the condition is met then the contents of the loop are performed. As with `for` loops, care must be taken to ensure that the loop will eventually end. That means that the condition must contain a variable that gets changed in the loop body.

Example 4.2

The second loop will almost certainly produce unexpected results. What seems to be the problem with it?

```
#include<stdio.h>
char choice;
int  num;

num= 8;
while (num < 20)
{
  printf("We're bad!\n");
  num+= 2;
}

while (choice == 'Y')   /* got a problem here */
{
  printf("Do you want to continue looping? ");
  scanf("%c",&choice);
}
```

4.3 `do-while` loops

The basic form of `do-while` loops is as follows:

`do` *loop-body* `while (`*condition*`)`

The semantics are exactly the same as with the `while` loop except that the condition is checked at the bottom of the loop instead of the top.

Question

How could the last while loop in example 4.2 be changed to be a `do-while` loop to fix the problem?

5 Data Structures

Data structures are necessary for all but the simplest of computer programs. This section discusses arrays and structures (records). Pointers have been included in this section more for convenience of presentation than for taxonomical purity.

5.1 Arrays

An array is a list of contiguous variables all having the same type. This example illustrates common uses:

Example

The values in brackets specify *how many* variables of the given type are to be reserved. Arrays in C are always indexed with integers starting at 0. Strings in C are simply arrays of characters. A string is terminated with a binary zero, so the most number of letters `name` could store below is 79 (plus the 0 character). Notice that strings and arrays of numbers are handled differently.

```
#include<stdio.h>
#include<string.h>     /* necessary for strlen */
int    a[100],i,n;
char   name[80];
double values[1000],x;

for (i=0; i<100; i++)  /* Does this get ALL of them? */
  a[i]= 0;

printf("Enter your name: ");
scanf("%s",name);            /* no & needed */
printf("%s, your name has %d characters\n",name,strlen(name));

/* lets assume that values contains n pieces of data that have
   already been initialized.  We'll find the average.          */
x= 0.0;
for (i=0; i<n; i++)
  x+= values[i];
printf("Average: %4.2f\n", x / (double) n);
```

5.2 Structures

While arrays allow storage of many items of the same type (under a single name), structures (or records) allow storage of items of different types under the same name.

Example

Each "piece" (field) of the structure is given a name and a type. Each field is then referenced by the variable name, a period, and the field name. A field can be a simple variable or an array. It is also possible to have an array of structures, as in the case of `everyone`. In this code segment, `jack` is initialized to be 35 units old with an id number of 12232. (His name, is, of course, Jack). `fred` is twice as old (and apparently joined the entity in question twice as late). `everyone` is either a replica of `fred` or `jack`. Which elements of `everyone` match `jack`?

```
#include<string.h>
struct personInfo
{
  int  id;
  char name[80];
  int  age;
} jack,fred,everyone[100];

jack.id= 12232;
strcpy(jack.name,"Jack");
jack.age= 35;

fred.id= 2*jack.id;
strcpy(fred.name,"Fred");
fred.age= 2*jack.age;

for (i=0; i<100; i++)         /* do we need {}'s? */
  if (i % 2)
  {
    everyone[i].id= jack.id;
    strcpy(everyone[i].name,"Jack");
    everyone[i].age= jack.age;
  }
  else
  {
```

```
    everyone[i].id= fred.id;
    strcpy(everyone[i].name,"Fred");
    everyone[i].age=
    fred.age;
  }
```

5.3 Pointers

A pointer is a 32-bit variable that contains an address. Every location in a computer's memory has an address. Computers are typically byte-addressable, so every byte of memory has its own unique address. We have already seen the ampersand operator (&) used to obtain the address of a variable for use with `scanf`.

All addresses are of the same size, although some variables obviously take up more room than others. At any rate, keeping track of the location of a big variable has the same expense (4 bytes) as keeping track of the location of a small variable.

The `*` symbol is used to specify a pointer variable during declaration. This can be a bit confusing since `*` is used to mean multiplication in some contexts! Consider:

Example

Here `aptr` and `bptr` are given the same value (the address of `a`). Printing an address (`%p` stands for "pointer") is about useless (except possibly for some debugging purposes), but as with many things in C, can be done. The `*` in the second `printf` takes on a *third* meaning: dereferencing. In that context, `*aptr` means "the value found at the address stored in `aptr`". What value for `*bptr` gets output in this example?

```
int a=2,b=3;
int *aptr, *bptr;

aptr= &a;
bptr= &a;   /* this is "= &a" on purpose */

printf("The address of a is %p\n",aptr);
printf("The value of a is %d\n",*aptr);
a= 7;
printf("The value at address in bptr is %d\n",*bptr);
```

There are two main reasons why one might want to use pointers: dynamic memory allocation and parameter passing. Parameter passing is discussed in section 6.2. For examples of dynamic memory allocation, keep reading!

Example

First notice the use of `typedef`. This is a matter of convenience.

Look at the pointer declarations for `jack`, `fred`, and `everyone`. All of them are declared to be pointers to `struct`s. In the case of `everyone` we have an array of pointers to `struct`s. To obtain memory in which to store the actual information, the `malloc` command can be used. The `sizeof` macro is used to determine how many bytes are necessary for storing the information. To use the new memory, we must dereference the pointer. The `.` operator is "stronger" than the `*` operator, so parentheses are required for grouping. This makes for rather cumbersome notation.

Pointers to `struct`s is a fairly common phenomenon, so it is not surprising that C has a shortcut notation: the `->` operator.

```
typedef struct personInfo PERSON;
struct personInfo
{
  int  id;
  char name[80];
  int  age;
};

PERSON *jack,*fred,*everyone[100];

jack= (PERSON *) malloc(sizeof(PERSON));
(*jack).id= 12232;
strcpy((*jack).name,"Jack");
(*jack).age= 35;

fred= (PERSON *) malloc(sizeof(PERSON));
fred->id= 12232;
strcpy(fred->name,"Fred");
fred->age= 35;

for (i=0; i<100; i++)
{
```

```
    everyone[i]= (PERSON *) malloc(sizeof(PERSON));
    everyone[i]->id= jack->id;
    strcpy(everyone[i]->name,"Jack");
    everyone[i]->age= jack->age;
  }
```

The natural question: WHY?!?! Suppose you may need to store anywhere from 1 to 100000 variables of type `PERSON` as in the previous example. It would be "wasteful" in terms of memory usage to have an array of 100000 variables, each of which require 88 bytes. Instead you can have an array of 100000 variables, each containing 4 bytes (i.e., pointers) and simply allocate the additional memory as needed.

It should also be mentioned that a number of data structures such as linked lists and trees are most intuitively represented using pointers as well.

Note

In the previous example we saw how to dynamically allocate memory at run-time. How do we *de*allocate memory? Assuming the same declarations above, to deallocate the memory allocated for `fred`, this statement will suffice: `free(fred)`.

6 Subprograms

6.1 General Information

Large programs that work are always split into smaller chunks. This provides organization and allows repetitive tasks to be invoked with a single line instead of having to cut-and-paste sections of code throughout the program. Consider a text-based program that constantly asks the user to "hit <enter> to continue". Since that request may be made in hundreds of places in a large program, it is a good candidate for creating a subprogram.

In C, subprograms are called functions. Before `main()`, a prototype of all functions must be given. The prototype describes what the interface of the function looks like. The keyword `void` preceding both the prototype and the actual function definition, specifies the return value of the function. In this case, the function doesn't return any information, so the return value is set to `void`.

Example

```
#include<stdio.h>

void hitReturn();  /* a "prototype" */

void main()
{
  /* do some stuff */
  hitReturn();

  /* do some other stuff */
  hitReturn();

  /* finish up everything */
  hitReturn();
}

void hitReturn()
{
  char result[3];
  printf("\n\n-----------------------------------------------\n");
  printf("Hit <Return> to Continue");
  fgets(result,3,stdin);
}
```

6.2 Parameters

Many subprograms behave differently based on the values of variables found in the main program. Values are communicated to a subprogram by "passing them as parameters." Consider:

Example

This subprogram has one parameter (a string). The value of the parameter passed to the function depends on the condition causing the error. Thus, the error message is customized to the error condition.

```
#include<stdio.h>
void reportError(char msg[]);

void main()
```

```
{
  /* some stuff */
  if (something_bad_happens)
    reportError("Something Bad Happened!");
  if (something_awful_happens)
    reportError("Something Awful Happened!");
  if (running_Windows98)
    reportError("Crashed for Apparently no Reason!");
  .
  .
  .
}

void reportError(char msg[])
{
  printf("ERROR: %s\n",msg);
  abort();    /* exits program immediately/ugly */
}
```

Another Example

This extended example demonstrates several concepts such as the use of local variables, declaring functions that return values, and passing addresses (pointers) as parameters. For examples demonstrating the role of pointers as parameters see section 7.3.

Function `calcAvg` returns the average age of `n` people as a `double`. The function `getPersonInfo` allows the user to input a name an age for a person. It also updates the value of `n` and assigns and id number based on the order of entry. It is necessary to pass the *address* of `n` in this case because, by default C parameters are passed by value. That is, the parameter is a copy of the original. When passing an address, the parameter is a copy of an address. By dereferencing the address (copy), the original data is accessible.

```
#include<stdio.h>

typedef struct personInfo PERSON;
struct personInfo
{
  int  id;
  char name[80];
```

```
  int  age;
};

double calcAvg(PERSON *[],int);
void getPersonInfo(PERSON *, int *);

void main()
{
  PERSON *people[100];
  int    i,n;
  double avg;

  n= 0;
  for (i=0; i<3; i++)
  {
    people[i]= (PERSON *) malloc(sizeof(PERSON));
    getPersonInfo(people[i],&n);
  }

  avg= calcAvg(people,n);
  printf("Average age is: %3.1f\n",avg);
}

double calcAvg(PERSON *everyone[],int n)
{
  int    i;
  double tot;

  tot= 0.0;
  for (i=0; i<n; i++)
    tot+= (double) everyone[i]->age;
  return tot / (double) n;
}

void getPersonInfo(PERSON *p, int *n)
{
  printf("Enter age: ");
  scanf("%d",&(p->age));
  printf("Enter name: ");
  scanf("%s",p->name);
```

```
    (*n)++;
    p->id= 10000+(*n);
  }
```

Note

Array names are, in fact, pointers. Therefore, when an array is passed as a parameter, the address of the array is made available to the subprogram. Therefore, it is *NOT necessary* to explicitly pass the address of an array.

Part II
C++

7 Introduction

Most of what there is to be known about C++ can be summed up in a single statement: C++ is a superset of C. There are some significant enhancements provided by C++, however.

7.1 Comments

One minor, but handy addition to C provided by C++ is the ability to do end-of-the-line comments with `//`.

Example

This methods of commenting is more convenient in many cases and can be nested within the C-style `/* */` comments.

```
int age;      // the age of the monkey
int IQ;       // the IQ of the monkey
```

7.2 Basic I/O

Input and output in C++ is typically done using the `cin` and `cout` objects provided in the `iostream.h` header file. These can be used in lieu of `printf` and `scanf`.

Example

The `cout` object is used for output and the `cin` object is used for input. Notice that it is not necessary to tell these objects what kind of data are being used ... they know how to handle it properly. The identifier `endl` produces the newline character.

```
#include <iostream.h>

void main()
{
  char name[80];
```

```
  cout << "I want to see some output!\n";
  cout << "What is your name: ";
  cin >> name;
  cout << "Good to see you " << name << "!" << endl;
}
```

7.3 Parameter Passing with &

The discussion of pointers and parameters can be illustrated by the following set of examples. The first example demonstrates a program that doesn't work properly. The next example fixes the problem using standard C notation. The third example demonstrates how the program can be written in C++ without the disturbing side-effect of using confusing notation.

Example (Doesn't Work)

In this program, the program presumably wants to have the user enter a value and store that number (plus one) into the parameter that was passed. C always passes parameters by value[1] so the changes made to `num` in `getNum` are being made to a copy of the parameter `n`. When the value for `n` gets printed, there is no change.

```
#include<stdio.h>
void getNum(int);

void main()
{
  int n;
  getNum(n);
  printf("Num: %d\n",n);
}

void getNum(int num)
{
```

[1]As mentioned earlier, this is technically true, although, it appears to the false in the case of arrays. An array name *is* an address, so passing it as a parameter causes the address of the array to be passed (by value). Of course, references to the array result in changes to the original array even though it is being reference through a copy of the address.

```
  printf("Enter value: ");
  scanf("%d",&num);
  num++;
}
```

Example (C Fix)

The problem with the first example is fixed by passing the *address* of `n` as a parameter. Thus changes are made to the original value. The "problem" with this method is not that it doesn't work, but simply that the change of notation within the function `getNum` is somewhat annoying.

```
#include<stdio.h>
void getNum(int *);

void main()
{
  int n;
  getNum(&n);
  printf("Num: %d\n",n);
}

void getNum(int *num)
{
  printf("Enter value: ");
  scanf("%d",num);
  (*num)++;
}
```

Example (C++ Fix)

This version of the program allows the programmer to keep the same notation as in the original example, but still passes the parameter by reference, causing the program to perform as intended. That is, the ampersand (&) is simply a flag in the parameter list that tells the compiler to pass the parameter by reference (variable) instead of by value.

```
#include<stdio.h>
void getNum(int &);
```

```
void main()
{
  int n;
  getNum(n);
  printf("Num: %d\n",n);
}

void getNum(int &num)
{
  printf("Enter value: ");
  scanf("%d",&num);
  num++;
}
```

Note

When passing a large structure (or an object as we will see in section 8), it is often desireable to pass the address so that it is not necessary to copy the entire variable on each function call. Of course, passing a parameter by reference implies that the variable will be modified by the subprogram. To prevent modification of the parameter while maintaining the efficiency of passing by reference, use the keyword const in the function declaration as follows:

```
void myfunc(bigType const &bigVar)
    .
    .
    .
```

8 Classes

The *big* change from C to C++ is the addition of classes and object-orientation. Object-oriented programming proponents would blanch at this statement: C++ classes can be understood and used effectively by viewing them as C structures with additional functionality.

A structure allows the programmer to combine variables of different types into a single big variable. A class allows the programmer to combine variables of different types *and* functions into a single big variable called an object.

8.1 Terminology

Here are some terms to ponder when the rest of the examples in this section don't make sense.

class	a declaration of a special variable that can combine variables and functions together in a single big variable
object	an instance of a class (i.e., an actual variable, not just a type definition)
member function	a function that is part of a class
inheritance	variables and/or functions that are passed from one class to another

8.2 Classes and Objects

The following extended example implements a program that creates a class called `PlainDie` (where die is singular for dice). A die is an "object" that has a current value (the side facing up), that has 6 sides, and that can be "rolled" to produce a (possibly) new value.

Example

This example shows the class declarations necessary to produce a "plain" 6-sided die. Each class has a constructor and a destructor. The constructor is a function that is called each time an instance of the class is created. Conversely, the destructor is a function that is called each time an instance of the class is destroyed. Neither of these special functions has a return value. The constructor always bears the name of the class and the destructor always bears the name of the class preceded by the tilde.

The variable `currentValue` is declared as a protected integer. "Protected" means that it cannot be accessed from outside the class. That is, the internal functions `roll` and `value` are the only functions that are allowed to access its value. This prevents a rogue programmer from assigning an invalid value to the die. It is the job of the function `value` to return the value of `currentValue` so that other parts of the program can view it. The `roll` function is currently the only entity besides the constructor that changes the value of `currentValue`. It uses the `rand` function found in `stdlib.h` to assign a value in the proper range.

`PlainDie` is a class and `die1` and `die2` are instances of the class (i.e., objects). Note the "structure-like" notation used to access the functions of each object. What do you think would happen if the following line were inserted into the program right before the `for` loop: `cout << die1.currentValue;`?

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class PlainDie
{
  public:
    PlainDie();
    ~PlainDie();
    void roll();
    int value() { return currentValue; };

  protected:
    int currentValue;
};


void main()
{
  int i;
  PlainDie die1,die2;

  for (i=0; i<10; i++)
  {
    cout << "Die 1: " << die1.value();
    cout << "\t Die 2: " << die2.value() << "\n";
    die1.roll();
    die2.roll();
  }
}

PlainDie::PlainDie()
{
   srand( (unsigned)time( NULL ) );
   roll();
}

PlainDie::~PlainDie()
{
}
```

```
void PlainDie::roll()
{
  currentValue= rand() % 6 + 1;
}
```

The output of this program is as follows:

```
Die 1: 4        Die 2: 4
Die 1: 4        Die 2: 1
Die 1: 4        Die 2: 1
Die 1: 6        Die 2: 1
Die 1: 6        Die 2: 3
Die 1: 5        Die 2: 1
Die 1: 6        Die 2: 3
Die 1: 3        Die 2: 4
Die 1: 3        Die 2: 3
Die 1: 1        Die 2: 3
```

8.3 Inheritance

Suppose a cavalier programmer decides to invent a multi-sided die. The "old" method of changing things would lead the programmer to change the definition of `PlainDie` to include a variable `numberOfSides` that tells how many sides a particular die has. It would also require the changing of the definition of `roll`. One problem with this method is that all the code that uses the current definition of `PlainDie` would have to be rewritten. Also, it seems a shame to have the overhead of generality when almost all the dice will have 6 sides anyway. Inheritance to the rescue.

Example

This example demonstrates the use of inheritance to produce an enhanced class called `Die`. This class inherits all of the functionality of `PlainDie`, but overrides the definition of `roll` and adds a new protected variable called `numberOfSides`. Notice that the definition of `PlainDie` is unchanged.

This example also demonstrates that one can use pointers to objects much in the same way as pointers to structures. One difference is the C++ `new` operator that makes allocation much simpler.

Notice that the constructor for `Die` calls the constructor for `PlainDie`. That is included in this example to demonstrate that it can be done. Usually that is a good way simplify a constructor. One problem with doing that in this case is that the `roll` function belonging to `PlainDie` gets called at initialization instead of of the newly defined `roll` function. To compensate for this `roll` gets called again in the `Die` constructor.

Another flaw with this method is that placing the `srand` command inside the constructor means that all die objects created within a given second will have the same initial value generated.

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class PlainDie    // same as before
{
  public:
    PlainDie();
    ~PlainDie();
    void roll();
    int value() { return currentValue; };

  protected:
    int currentValue;
};

class Die : public PlainDie  // inherits from PlainDie
{
  public:
    Die(int = 6);            // 6 sides by default
    ~Die();
    void roll();
    int sides() { return numberOfSides; };

  protected:
    int numberOfSides;
};

void main()
{
```

```
  int i;
  PlainDie die1;
  Die *die2;
  Die die3(12);

  die2= new Die(8);
  cout << "Die #1 has 6 sides\n";
  cout << "Die #2 has " << die2->sides() << " sides\n";
  cout << "Die #3 has " << die3.sides() << " sides\n";
  cout << "-----------------------------------------\n";

  for (i=0; i<10; i++)
  {
    cout << "Die 1: " << die1.value();
    cout << "\t Die 2: " << die2->value();
    cout << "\t Die 3: " << die3.value() << "\n";
    die1.roll();
    die2->roll();
    die3.roll();
  }
  delete die2;
}

PlainDie::PlainDie()
{
  srand( (unsigned)time( NULL ) );
  roll();
}

PlainDie::~PlainDie() { }

Die::Die(int numSides) : PlainDie()
{
  numberOfSides= numSides;
  roll();
}

Die::~Die() { }

void PlainDie::roll()
```

```
{
  currentValue= rand() % 6 + 1;
}

void Die::roll()
{
  currentValue= rand() % numberOfSides + 1;
}
```

The output of this program is as follows:

```
Die #1 has 6 sides
Die #2 has 8 sides
Die #3 has 12 sides
------------------------------------------
Die 1: 1          Die 2: 2        Die 3: 10
Die 1: 5          Die 2: 8        Die 3: 10
Die 1: 1          Die 2: 3        Die 3: 1
Die 1: 6          Die 2: 8        Die 3: 3
Die 1: 1          Die 2: 4        Die 3: 11
Die 1: 4          Die 2: 2        Die 3: 5
Die 1: 2          Die 2: 3        Die 3: 1
Die 1: 1          Die 2: 2        Die 3: 12
Die 1: 5          Die 2: 5        Die 3: 5
Die 1: 4          Die 2: 2        Die 3: 3
```

9 Operator Overloading

Operator overloading is one of the more fascinating features of C++. It allows the programmer to redefine operators for use with programmer-defined data types. Though the examples given here are trivial, they demonstrate how to achieve operator overloading.

Suppose you have two dice `die1` and `die2` and you need to store the sum of their values into a variable called `sum`. This could be accomplished easily as follows:

```
sum= die1.value() + die2.value();
```

Of course, if you are constantly adding values of dice together in your program it would be nicer (and perhaps more intuitive) to simply say:

```
sum= die1 + die2;
```

For this method to work properly, you will have to redefine the meaning of the "+" operator when used with dice.

Example (Overloading + and ==)

In this example we show how to add values of dice with "+" and how to compare values of dice with "==". This example assume the same definition of `PlainDie` as in section 8.3 with the following additions:

```
class PlainDie
{
  public:
    int operator+(PlainDie die)
        { return currentValue + die.value(); };
    int operator==(PlainDie die)
        { return currentValue==die.value(); };
          .
          .
          .
}

void main()
{
  Die die1,die2;
  sum= die1+die2;
    .
    .
    .
}
```

Another kind of operator overloading can take the form of the so-called "stream insertion operator" (i.e., "¡¡"). If there is some standard way you want an object to be displayed, you can overload this operator so that it will produce output in the proper form without having to redo it every time.

Example (Overloading ¡¡)

Suppose you want to have the statement: `cout << die1;` to produce the following output: `Sides:  6, Value:  4` (where 6 and 4 are `numberOfSides` and `currentValue`, respectively. First overload the "¡¡" operator and then you're in business.

This example also demonstrates the use of a "friend" function. A friend function is declared outside of the class, but is allowed to access protected members of that class. In this example we assume that the definition of `Die` is the same as in section 8.3 except for the following addition:

```
class Die
{
  public:
    friend ostream &operator<<(ostream &, Die);
     .
     .
     .
}

void main()
{
  Die mydie;
  cout << mydie;
}

ostream &operator<<(ostream &output, Die die)
{
  output << "Sides: " << die.sides();
  output << ", Value: " << die.value() << endl;
}
```