C++ Tutorial

CHRISTOPHER G. HEALEY
North Carolina State University

1 BASIC PROGRAM STRUCTURE

A C++ program\(^1\) is made up of a mainline procedure called `main()`, and zero or more additional procedures to perform operations within your program. When a program starts execution, it begins by calling the mainline procedure.

Source code for a program can be contained in a single file, or split across multiple files. Additional functionality may also be provided via external libraries. The source code is compiled into object files. The object files are then linked together to produce an executable program.

2 OPERATORS

Every computer language provides a set of built-in functions or operators which allow you to perform simple operations such as addition, subtraction, increment, and decrement. C++ provides a useful set of operators to perform basic arithmetic, operator grouping, indirection, and binary logic. An expression is a combination of variables, constants, and operators. Every expression has a result. Operators in C++ have precedence associated with them; this means expressions using operators are not evaluated strictly left to right. Results from the operators with the highest precedence are computed first. Consider the following simple C++ expression:

\[ 6 + 3 \times 4 / 2 + 2 \]

If this were evaluated left to right, the result would be 20. However, since multiplication and division have a higher precedence than addition in C++, the result returned is 14, computed as follows.

- \( 3 \times 4 \rightarrow 12 \)
- \( 12 / 2 \rightarrow 6 \)
- \( 6 + 6 \rightarrow 12 \)
- \( 12 + 2 \rightarrow 14 \)

Of course, we can use parentheses to force a result of 20, if that’s what we wanted, with the following expression:

\[ ((( 6 + 3 ) \times 4 ) / 2 ) + 2 \]

Below is a list of the basic operators in C++, along with an explanation of what they do. The operators are group according to precedence, from highest to lowest.

\(^1\)Throughout these notes, I will use C++ conventions for comments (// rather than /* and */, memory allocation (new and delete rather than malloc() and free()), printing (cout rather than printf), argument passing, and so on. I will not discuss any of the object-oriented aspects of C++, however. This material is left to the reader, and can be found in any textbook on C++.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>parentheses define the order in which groups of operators should be evaluated</td>
</tr>
<tr>
<td>++, --</td>
<td>pre-increment and pre-decrement</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>address of, deference</td>
</tr>
<tr>
<td>()</td>
<td>casting</td>
</tr>
<tr>
<td>-&gt;, ..</td>
<td>member selection</td>
</tr>
<tr>
<td>*, /, %</td>
<td>multiplication, division, integer division</td>
</tr>
<tr>
<td>+, -</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>bitwise shift</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>less than, less than or equal, greater than or equal, greater than</td>
</tr>
<tr>
<td>==, !=</td>
<td>equality, not equal</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td>bitwise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td>logical OR</td>
</tr>
<tr>
<td>++, --</td>
<td>post-increment and post-decrement</td>
</tr>
<tr>
<td>=, *=, /=, etc.</td>
<td>assignment operators</td>
</tr>
</tbody>
</table>

Note the difference between the pre-increment and post-increment operators. Pre-increment occurs before any part of an expression is evaluated. Post-increment occurs after the result of the expression has been returned. For example, what would the output be from the following lines of code?

```cpp
a = 1;
b = (a++) - 2 * a++;
cout << "a = " << a << "\n";
cout << "b = " << b << "\n";

n = 1;
m = (++n) - 2 * ++n;
cout << "n = " << n << "\n";
cout << "m = " << m << "\n";
```

If we compiled and ran this code, we would get the results

```
a = 3
b = -1
n = 3
m = -4
```

3 CONDITIONALS

C++ also provides a number of built-in conditional statements. This allows us to execute certain parts of our program based on some condition being either true or false. Unfortunately, in C++ there is no boolean type. A value of 0 is considered false, any non-zero value is considered true. A well known example of a conditional statement is the if-then-else conditional. If some expression evaluates to true, then execute one part of our code, else execute a different part. Below is a list of the conditional statements available in C++, along with examples of how to use them.
3.1 if-then-else

if-then-else works exactly as you would expect. If some condition is true, the then portion of the conditional is executed, otherwise the else part of the conditional is executed. The else part of the conditional is optional. If it doesn’t exist and the expression is false, nothing is executed. Consider the following example, which sets a student’s letter grade based on their overall percentage.

```c
if ( n_grade >= 80 ) {
    l_grade = 'A';
} else if ( n_grade >= 65 ) {
    l_grade = 'B';
} else if ( n_grade >= 50 ) {
    l_grade = 'C';
} else {
    l_grade = 'F';
}
```

The braces are optional here, since the then and else code sections consist of only a single statement. If there were more than one statement, however, the braces would be required.

3.2 for

for is a looping conditional. It executes a set of statements repeatedly a specific number of times, then exits the loop. The condition here is a value which determines the number of times the loop will be executed. Often, a variable’s value is used for this purpose, as in the following example.

```c
cout << "Please enter a number:
";
 cin >> loop-count;
for( i = 0; i < loop-count; i++ ) {
    cout << "Loop pass " << i << "\n";
}
```

Notice the for statement is made up of three separate parts: an initialization part, a conditional part, and an interaction part.

```c
for( init-part; condition-part; iterate-part ) {
    ...
}
```

The initialization part is executed before looping begins. It is usually used to initialize variable values. In our example, we use it to set the value of i to 0. The condition part is checked each time the loop begins executing. If the condition is true, the loop is executed. If it is false, looping ends. In our example, we continue looping until i is greater than or equal to the value of loop-count. The interaction part is executed each time the loop finishes. It is usually used to update the value of a variable being used inside the loop. In our example, we increment i by 1 each time through the loop. Thus, i is used to count the number of times the loop has been executed.
3.3 while

**while** is also a looping conditional. While some condition is true, continue to execute a group of statements. The key feature of while is that it checks its condition at the top of the loop. This means the loop could be executed 0 times, if the condition is initially false. Consider the following example, which prints numbers in an array until a terminator value is encountered.

```cpp
i = 0;
while( a[ i ] != -1 ) {
    cout << "Index " << i << " has value " << a[ i ] << "\n";
    i++;
}
```

3.4 do-while

**do-while** is similar to the while conditional. While some condition is true, continue to execute a group of statements. Unlike while, do-while checks its condition at the bottom of the loop. This means the loop executes at least once, even if the condition is initially false. Consider the following example, which adds numbers to an array until the user enters a terminator.

```cpp
i = 0;
do {
    cout << "Enter a value (-1 to end):\n";
    cin >> in-val;
    a[ i++ ] = in-val;
}while( in-val != -1 );
```

3.5 switch

**switch** allows us to execute a piece of code based on a variable’s value. In essence, it says if the variable’s value is A, do this, if the variable’s value is B, do this, and so on. This is similar to a large if-then-else condition, but since switch statements can be optimized by the compiler, they are often more efficient to use and easier to understand. Consider the following example, which prints a message based on a student’s letter grade.

```cpp
switch( l_grade ) {
case 'A':
    cout << "This student got a great mark\n";
    break;
case 'B':
case 'C':
    cout << "This student got a good mark\n";
    break;
case 'D':
    cout << "This student barely made it\n";
    break;
```
The `switch` command specifies the variable whose value we want to compare against. In our case, we are switching on the value of `grade`. Each `case` statement specifies a possible value for the variable. If the variable’s value is equal to the case value, statements corresponding to the case condition are executed. So, if a student’s grade is A, we print the text “This student got a great mark” on the screen. If the student’s grade is B or C, we print “This student got a good mark”, and so on. If the variable doesn’t match any of the case values, the default portion of the switch statement is executed.

There are a few caveats to remember about switch statements. First, the variable we switch on must be numeric (float, int, double) or character. It cannot be a more complicated value like a string. Second, we need to include a `break` command at the end of each case condition. If we don’t do this, the case statement and all case statements following it will be executed.

3.6 break and continue

`break` and `continue` are two special commands provide by C to help us manage execution of conditional statements. `break` means “break out of” or leave the current conditional structure immediately, with checking the value of the condition. `continue` means jump immediately to the condition and test it. If it is still true, continue execution from the top of the conditional structure, otherwise quit.

4 VARIABLES

Variables in C++ are user-defined names which are used to maintain different values at different places in a program while it runs. You should remember that every variable maintains its value at some (compiler-chosen) location in memory. This memory location is the variable’s address. You can ask the program to return a variable’s address by using the `&` (address of) operator. Variables also have two related characters, extent and scope. A variable’s extent describes when a variable starts its life, and when it ends its life. A variable’s scope describes which parts of the program can see (and therefore access or modify) the variable.

4.1 Built-In Types

C++ supports built-in data types integer (int), floating point (float and double), and character (char), plus arrays of the same types. A variable is defined as a type followed by a name followed by an optional initialization:

```cpp
int i;
float j = 2.5;
char c[ 256 ];
double d[ 4 ][ 5 ];
```
4.2 Pointers

Pointers are key to understanding C++. Without a good grasp of pointers, you will most likely be in a constant state of amazement (and grief) as C++ does seemingly incomprehensible things. The good news is that pointers are easy to understand (in spite of what many people might tell you). K&R defines pointers as:

... a variable that contains the address of another variable

There are two important rules to remember about pointers. These relate to a pointer’s value and a pointer’s type. If you remember these two simple rules, then you understand what a pointer is.

1. A pointer’s value is a number, which represents a location in memory (an address) where information is stored.
2. A pointer’s type tells you the kind of information stored at the pointer’s address.

There is no magic or trickery associated with pointers. They’re simply a variable containing a number which represents the memory location (address) were a particular type of data is stored. There are three steps you should follow when you are using pointers.

1. Declare the pointer.
2. Initialize the pointer.
3. Dereference the pointer to obtain access to the information it points to.

A pointer is declared by placing an asterisk (*) between the pointer’s type and its name:

```c
int * i_ptr; // A pointer to an integer
char * c_ptr; // A pointer to a character
```

Any variable’s address can be obtained by preceding the variable’s name with the address of (&) operator. To dereference a pointer and retrieve the data value it points to, precede the pointer variable’s name with the dereference (*) operator.

It is very important to remember to initialize your pointers before you try to dereference them. Dereferencing uninitialized pointers will randomly destroy various areas of memory. Uninitialized pointers are one of the most frequent errors. Unfortunately, they are often the most difficult to track down, as well. Consider the small code fragment:

```c
int * i_ptr;
int x,y; // Two temporary integer values

x = 10; // Assign 10 to x
i_ptr = &x; // Initialize pointer
y = *i_ptr; // Dereference pointer
```

The statement `i_ptr = &x` assigns the address of `x` to `i_ptr`. The statement `y = *i_ptr` retrieves the value of the variable `i_ptr` points to (in this case `x`), and assigns it to `y`. The above code fragment is equivalent to the statement

```
y = x;
```
Consider the following diagram:

![Diagram](image)

The statement `i_ptr = &x` assigned the value of 2000 to `i_ptr`, since `x` is located at address 2000 in memory. The statement `y = *i_ptr` took the address from `i_ptr`, in this case 2000, and looked at that location in memory to see what value was stored there. It retrieved the value of 10, and assigned it to `y`.

### 4.3 Arrays

Arrays have a number of specific properties that you should be aware of:

1. Arrays index starting at 0. So, if you define a 256-character array `char c[256]`, its entries would be accessed as `c[0]` through `c[255]`.

2. If you store a string value in a character array, C++ needs to know where the string ends within the array. It does this by appending a null character (a character with an ASCII value of 0, defined as `'\0' after the last character in the string. This means that an array holding a string needs to be big enough to store the string plus the additional end-of-string marker. For example, this code snippet is incorrect:

   ```cpp
   char c[5] = "Hello"; // Array too small for 5 chars plus end-of-string
   ```

3. C++ makes no attempt to validate the indices you pass to an array. If you try to access an invalid array position, C++ will not inform you that the error has occurred (instead, your program will most likely crash mysteriously at some later point in its execution). For example, this code snippet is incorrect:

   ```cpp
   char c[5];
   c[5] = 'a'; // Array indexes 0..4, so there is no entry 5
   ```

4. If a procedure receives an array as an argument, it defines the argument as either a pointer to the array type, or as an array with an undefined size:

   ```cpp
   void proc_0( int * a ) // Argument is an integer array
   void proc_1( int a[] ) // Argument is an integer array
   ```

5. The name of an array is a special case. If you give the name of an array without any index, C++ returns the address of the first entry of the array:

   ```cpp
   char c[5]; // Character array
   char * s;  // Pointer to character data
   s = c;     // Assign address of c[0] to s
   ```

   This explains why the normal method of passing an array to a procedure (including how we define that we are expecting an array in the procedure header) works:

   ```cpp
   void proc( char * arr, int n )
   // arr: Array of one of more chars
   // n: Size of array
   ```
4.4 Scope

There are four types of scope in C++. Scope defines where in a program you can access or modify a given variable:

1. **Internal** or **local**, a local variable is known only inside the procedure or block in which it is declared. Most variable declarations are local, for example:

   ```
   {  
       int x, y;  
       ...  
   }
   ```

2. **External** or **global**, a variable which is declared outside of any procedure or block is an external variable, and is known throughout the source file in which it is declared. If an external variable is used in a source file other than the one where it is declared, an external declaration of the form

   ```
   extern type name;
   ```

   is necessary in the importing file.

3. **File**, a variable declared **static** outside of any procedure or block is a file variable, and is known throughout the source file in which it was declared. A file variable cannot be imported by other files using `extern`.

4. **Struct**, the scope of a variable which is a member of a structure is that structure (i.e., a variable with the same name declared outside the structure is a different variable).

4.5 Extent

There are three types of extent in C++. Extent defines when a variable is created, and when it is destroyed:

1. **Automatic**, an automatic variable is created whenever the procedure or block in which it is declared is executed; it is destroyed whenever the procedure or block is exited.

2. **Dynamic**, a dynamic variable has its memory explicitly allocated and destroyed using operators `malloc()` and `free()` (in C) or `new` and `delete` (in C++)

3. **Static**, a static variable is created when the program is initially loaded and executed; it retains its value until the end of the program (i.e., it is global to the entire program). Remember, as noted above, the scope of a static variable cannot exceed the source file in which it is declared.

As an example of the ideas of scope and extent, consider the following code fragment, and the corresponding table, which describes the scope and extent of each variable declared in the code fragment.
Table 1: A list of the variables from the previous program, showing their extent and scope; boxes in the table labeled — denote an extent-scope pair which is not possible

<table>
<thead>
<tr>
<th>Extent</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>a, c, w, k (in main)</td>
</tr>
<tr>
<td>static</td>
<td>b</td>
</tr>
<tr>
<td>dynamic</td>
<td>integer referenced by w</td>
</tr>
<tr>
<td>file</td>
<td>i</td>
</tr>
<tr>
<td>external</td>
<td>—</td>
</tr>
</tbody>
</table>

5 DYNAMIC MEMORY ALLOCATION

By default, when you declare a variable C++ allocates spaces to hold its value, then binds its name to the chosen address. When you use the address of (&) operator on the variable’s name, this is the address that’s returned. The allocating and freeing of the variable’s memory is performed automatically for you, based on the variable’s scope.

Sometimes, however, we need more explicit control over when memory is allocated, and when it is returned to the system. This can be accomplished using the dynamic memory operators new and delete.

new allocates a block of memory from the heap, returning the address of the block to the caller. new requires you to specify both the type of value to be stored in the block, and optionally the number of values to be stored (if you need create an array). The address must be saved in an appropriate variable. For example:

```cpp
cchar * c_ptr;
int * i_ptr;
```
c_ptr = new char;       // Dynamically allocate one char
i_ptr = new int[20];   // Dynamically allocate array of 20 ints

Memory allocated with \texttt{new} stays allocated until you explicitly deallocate it, or until the program ends. You can free up memory using the \texttt{delete} operator. \texttt{delete} requires you to provide a pointer to a block of memory previously allocated with \texttt{new}, for example:

\begin{verbatim}
delete c_ptr;          // Free dynamically allocated char
delete [] i_ptr;      // Free dynamically allocated array of ints
\end{verbatim}

Notice that standard C++ convention is to put square braces ([ and ]) between the \texttt{delete} keyword and the pointer variable’s name if the block to be deleted is an array\footnote{In this example, even if [ ] is not included, the deallocation will function correctly; however, if you are deleting an array of objects that have destructors associated with them, [ ] is required to force C++ to walk the array and invoke each object’s destructor properly.}.

Along with greater flexibility comes greater responsibility. Your program is required to keep track of all memory it allocates, to ensure it is accessed properly, and to ensure it is freed properly. Improper use of dynamic memory allocation is one of the main causes of runtime errors in a program. Here are a few tips (together with examples) of things to remember when performing dynamic memory allocation:

1. Never reference a block of memory after it has been freed. For example:

\begin{verbatim}
int * i_ptr;
int * j_ptr;
i_ptr = new int;
j_ptr = i_ptr;
*i_ptr = 10;
...
delete i_ptr;
cout << "The value in the memory block is " << *j_ptr << "\n";
\end{verbatim}

An attempt is made to reference the memory block via \texttt{j_ptr} in the \texttt{cout} statement, but the memory \texttt{j_ptr} points to has already been freed (in the statement \texttt{delete i_ptr;}). The result of running this code snippet is random; it may work, or it may not.

2. Always have at least one pointer variable maintaining the address of each memory block you allocate. The following code snippet demonstrates an example of this need:

\begin{verbatim}
void proc( void )
{
    int * i_ptr;
    i_ptr = new int[10];
}
\end{verbatim}

A block of memory is allocated within \texttt{proc()}, and its address is assigned to \texttt{i_ptr}. When we leave \texttt{proc()}, all of its local variables, including \texttt{i_ptr}, are automatically destroyed. At this point, we no longer have an active pointer variable with the address of the integer array we allocated. The integer array has been lost (its memory has leaked). It will stay allocated until the program ends, but will be unavailable for use. A proper definition for the procedure would be something like:
int * proc( void )
{
    int * i_ptr;
    i_ptr = new int[ 10 ];
    return i_ptr;
}

int * j_ptr;
j_ptr = proc();

This passes the address out of the procedure, and stores it in an active integer pointer variable.

3. Just as with arrays, you are responsible for ensuring that you do not try to access outside the bounds of the memory block, for example:

int * i_ptr;
i_ptr = new int[ 5 ];
i_ptr[ 5 ] = 10; // Error: only 5 entries, i_ptr indexes 0..4

Again, C++ will not generate an error message if you try to access outside a dynamically allocated block of memory.

6 PROCEDURES

As with all programming languages, procedures are used to divide a program into logical units. Procedures also promote code reuse for common processing within a program. A procedure is defined as a return type, a procedure name, and zero or more arguments (each with their own type and name) expected by the procedure. For example:

float cube( float v );
void display( void );

Notice that if the procedure returns no value, its return type is normally defined as void. Similarly, if a procedure expects no arguments, its argument list is defined as void. Some compilers allow both void keywords to be left out (e.g., display()); rather than void display( void );), although the C++ standard requires a return type be present for every procedure. Regardless, it is often informative to explicitly define return codes and argument lists.

6.1 Argument Passing

By default in C++, parameters passed to a procedure cannot have their value changed by the procedure. This means that no matter what you do to the parameter inside the procedure, the original variable’s value never changes. Consider an example of this, a simple swap procedure which is designed to swap two numbers.

void swap( int x, int y )
{
    int temp;
    temp = x;
```c
x = y;
y = temp;
}

int main()
{
    int i = 10;
    int j = 15;

    swap( i, j );
    cout << "i = " << i << "\n";
    cout << "j = " << j << "\n";
}

If we compile and run this program, we will get back the result:

    i = 10
    j = 15

That is, the values of the variables \( i \) and \( j \) weren’t swapped at all. A procedure cannot permanently change the value of arguments it receives.

The swap procedure shown above uses something called pass by value. In simple terms, pass by value means “pass a copy of the variable, do all work using the copy, and make no changes to the original variable”. Of course, sometimes want to pass parameters to a procedure and have the procedure permanently change the value of those parameters. This is called pass by reference. Pass by reference means “pass a reference to the variable, do all work using the reference to the variable, and thus all changes are made directly to the original variable”. You can choose pass by reference for a procedure’s argument in C++ list by putting an ampersand between an argument’s type and its name. For example, to make the swap procedure use pass by reference, replace it’s definition with:

```c
void swap( int & x, int & y )
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

The swap procedure will now swap the values of the two original parameters, which is want we wanted.

How does C++ do pass by reference? In fact, it uses pointers to get at the memory location where each argument’s value is stored. Changes are made directly to that memory location, and the variable’s value is modified as a result. We can “simulate” pass by reference by passing addresses to the swap procedure, as follows.

```c
void swap( int * x, int * y )
{
    int temp;
```}

3In C, there is no shorthand to denote pass by reference; you must explicitly use pointers to pass a reference to an argument.
```c
int main()
{
    int i = 10;
    int j = 15;

    swap( &i, &j );
    cout << "i = " << i << "\n";
    cout << "j = " << j << "\n";
}
```

Executing this program would give the correct result, namely \( i = 15 \) and \( j = 10 \). How does this work? Note in the swap procedure we are not changing the value of the parameters. We passed the address of two integers to `swap()`, and those addresses never change. However, we do change the values located at those addresses. Consider the diagram:

After the statement `temp = *x`, we have:

After the statement `*x = *y`, we have:

Finally, after the statement `*y = temp`, we have:
Note that the parameters $x$ and $y$, which contain the address of $i$ and $j$, never changed. However, the values of $i$ and $j$ have been swapped, which is what we wanted.

7 C++ FILES

Files that you write to build an executable program are divided into two types: header files and source code files. These files are normally paired, that is, each source code file `foo.cpp` has a corresponding header file `foo.h`. The source code file contains the implementation for a set of procedures. Usually, these procedures are usually related to one another (e.g., procedures to deal with date functions, or procedures to deal with time functions, or procedures that belong to a common C++ class). The corresponding header file defines the interface for all the procedures in the source code file you want to make publicly available to other parts of your program. It includes the names, argument lists, and return types of each procedure, along with other items like public `struct` definitions.

If some other source code file `bar.cpp` wants to call procedures in `foo.cpp`, all it needs to do is `#include "foo.h"`. This provides all the information `bar.cpp` needs to know which procedures `foo.cpp` provides, and how to invoke them (e.g., what their names are, what arguments they expect, and what type of value they return).

How does the compiler ensure that the procedures `foo.h` claims to provide are actually included as specified in `foo.cpp`? This happens during linking. For each procedure requested in any source code file, the compiler searches all the object files it is linking for an implementation for the given procedure. If an implementation is found, the procedure call is allowed. If an implementation is not found, a “missing procedure” link error is generated.

7.1 Header Files

Header files (or include files) are used to define a public interface into a C++ source code module. Usually, a C++ source code module is a collection of data types, procedures, and class definitions which provide some kind of service (i.e., an abstract data type). Other modules who need the service include this header file. The header file provides all the information necessary to use the module, including:

- a list of available procedures, including their names, the arguments they expect, and the type of value they return,
- definitions for any specialized data structures used by the module and its procedures,
- any constants or fixed values used by the module and its functions and methods, and
- `#include` statements for any special data types the header file needs to define to properly compile.

In C++ terms, things like constant definitions, structure definitions, class definitions, and procedure prototypes can all be put inside a header file. Since a header file defines a public interface, information which is private to the module must either be marked as private, or not included within the file. In general, a header file does not provide the following information:

- descriptions of how various procedures perform their jobs (that is, little or no C++ code is included in the header file)
• descriptions of or access to private variables which are used inside the module by the procedures to maintain state

As an example of a header file, consider the following which defines the public interface for a module which manipulates dates.

```c
// DATE.H
// Constants, structures, and class definitions for DATE.CPP

#ifndef _date_h
#define _date_h

const int NUM_OF_MONTH = 12; // Number of months in one year

struct date_S { // Date structure definition
  int day;
  int month;
  int year;
};

void print( date_S );
void set( date_S&, int, int, int );

#endif
```

This interface defines a single constant, NUM_OF_MONTH, a new structure date_S to hold date information, and a number of procedures which operate on dates.

If some other module, say employee.cpp, needs to perform date operations, it simply includes the header file by placing the command `#include "date.h"` at the top of employee.cpp. Now all the procedures inside employee.cpp know about the date structure and the methods which manipulate dates.

Finally, notice the compiler `#ifdef`, `#define`, and `#endif` directives included in the header file:

```c
#ifdef _date_h
#define _date_h
...
#endif
```

These are used to ensure the header file is never processed more than once during any single execution of the compiler. Basically, the directives say, “Has the variable _date_h been defined? If not, define it, then process the remainder of this file. If already defined, skip everything up to the `#endif` directive (that is, skip the remainder of the header file).”

What happens if a header file is processed more than once by the compiler? The compiler will complain that procedures are being multiply defined. That is, the compiler thinks you are trying to define two different procedures with exactly the same name, return type, and argument list. This is not allowed.

You might wonder how a header file could be `#include`d more than once for a given source code file. Remember, header files can themselves ask to `#include` other header files (in fact, this often happens, since one header file may need something defined in a different header file). This means that a single header file can often be `#include`d more than once during a common compile pass. The compiler directives ensure that the definitions in the header file will only be processed the first time, and will be ignored during all subsequent includes.
Note also that there is nothing special about the variable name used in header file to protect it (e.g., the _date.h used in this example). The only requirement is that every header file uses a _unique_ variable name to protect itself. The easiest way to do this is to define the variable name as a derivative of the header file’s name. Since two files with the same name cannot occur, two header files with the same variable name will never occur, either.

### 7.2 Source Code Files

For completeness, here is the source code implementation for these simple date operations:

```cpp
// DATE.CPP
// Implementation of simple date operators

#include <fstream.h>
#include "date.h"

static char * month_name( int m );

void print( date_S dt )
{
    cout << month_name( m[ dt.month ] );
    cout << " " << dt.day;
    cout << ", " << dt.year << "\n";
}

static char * month_name( int m )
{
    // Return month’s short name, or Unk for an unknown month
    // m: Numeric month, 1=January, 12=December

    char name[ 13 ][ 4 ] = {
        "Unk"
    };

    if ( m < 1 || m > 12 ) {
        return name[ 12 ];
    } else {
        return name[ m - 1 ];
    }
}

void set( date_S & dt, int d, int m, int y )
{
    dt.day = d;
    dt.month = m;
    dt.year = y;
}

Note that the procedure month_name() is _not_ public, so its prototype is not in the header file date.h. Instead, it is marked _static_ (to restrict its scope to the file in which it is defined), and both its prototype and its implementation
are located inside date.cpp. No one outside date.cpp can call month_name().

8 INPUT/OUTPUT

C++ supports all the various types of I/O you would expect, including reading from the keyboard, writing to the screen, and writing and reading to and from files stored on disk.

8.1 Keyboard and Screen Input/Output

C++ provides three built-in stream objects to read from standard input, write to standard output, and write to standard error: cin, cout, and cerr, respectively. They are made available by adding #include <iostream.h> at the top of a source code module that uses the objects.

Reading from the keyboard is normally done with cin and the input redirection operator >>, for example:

```cpp
#include <iostream.h>

int i;
float f;
char str[256];

cin >> i; // Read an integer value from the keyboard
cin >> f; // Read an floating point value
cin >> str; // Read a C++-type string
```

If you use redirection with cin, you must be careful to ensure the proper type of data is typed at the keyboard. cin will fail if either end-of-file is read, or if invalid input is seen. For example, consider the following code fragment:

```cpp
#include <iostream.h>

int i;
int sum = 0;

cin >> i;
while( i >= 0 ) {
    sum += i;
    cin >> i;
}
```

This program reads integer values until a negative value is entered. Suppose you typed a value of 3.14159. 3 would be read and assigned to i. The remaining input .14159 would be left on the input stream as not valid for integer input. The program would then loop forever, waiting for a valid integer but never seeing one (since .14159 remains “stuck” at the front of the input stream). You can detect this by checking the return code for the redirection operation. cin returns false if invalid input or end-of-file is seen:

```cpp
#include <iostream.h>

int i;
int sum = 0;

cin >> i;
while( i >= 0 ) {
    sum += i;
    cin >> i;
}
```

4 C has its own predefined file structs of type FILE * for standard input, standard output, and standard error named stdin, stdout, and stderr, respectively; they are accessed through #include <stdio.h>, and are invoked with functions like printf(), scanf(), and fprintf().
int i;
int sum = 0;

while( ( cin >> i ) && i >= 0 ) {
    sum += i;
}

Writing to standard output (i.e., to the screen) is normally done with cout and the output redirection operator <<, for example:

```cpp
#include <iostream.h>
int i;
int sum = 0;

cout << "Type an integer value (-1 to stop): ";
while( ( cin >> i ) && i >= 0 ) {
    sum += i;
}

cout << "\nThe sum of the values you typed is: " << sum << "\n";
```

Input and output redirection are defined for all built-in types (e.g., character, integer, float, double, C++-type string). During input redirection into a variable, characters typed at the keyboard are automatically converted into a value that is stored in the variable. During output redirection, the variable’s value is printed as a text string. Multiple variables or explicit strings can be chained together in a redirection statement by separating them with >> or <<. Finally, special characters like newline (\n) and tab (\t) can be used within explicit strings to allow you to format your output.

Writing to standard error works identically to writing to standard output, except that cerr is used rather than cout. Although both objects send text to the screen, they are in fact separate streams, with output that can be distinguished (e.g., with pipe and redirect operators at the UNIX command prompt). One important difference is that cout may temporarily buffer its text before it’s printed on the screen. cerr prints its text immediately when the redirection statement is executed. This can affect situations where you’re printing text messages to try to determine where your program is failing. With cout, you might miss some of the final messages, since they could be buffered and then lost when the program crashes. With cerr, however, you’re guaranteed to see all the messages up to the time the program fails.

### 8.2 Additional Input/Output Commands

C++ provides a number of additional methods that you can use to read or write data in different ways:

- **get( char & ch )**
  Extracts a single character from an input stream, stored in character variable ch. Returns a reference to the istream object on which it was invoked.

- **put( char ch )**
  Places the value of character variable ch to an output stream. Returns a reference to the ostream object on which it was invoked.
getline( char *buf, streamsize size, char delim="\n" )
Reads a stream of characters from an input stream and stores them in the character array buf. Input continues until: (1) the delimiter character delim is read; (2) end-of-file is read; or (3) size-1 characters are read. buf is automatically NULL-terminated to form a valid C++-type string. If a delimiter character is seen, it is removed from the input stream, but not included in buf. If no delimiter is specified, it defaults to newline. Returns a reference to the istream object on which it was invoked.

gcount()
Returns the number of characters read with the istream object’s most recent getline() or get().

read( char *buf, streamsize size )
Reads exactly size characters from an input stream, placing them in the character array buf. Returns a reference to the istream object on which it was invoked.

write( char *buf, streamsize size )
Writes exactly size characters to an output stream. Returns a reference to the ostream object on which it was invoked.

Since these are methods, they must be invoked through an appropriate object, for example, through cin or cout. Recall that in C++, methods are invoked on an object by specifying object_name.method_name, for example:

```cpp
char buf[ 256 ];
cin.getline( buf, 256 );
cout << cin.gcount() << " characters read...\n";
cout << buf << "\n"
```

8.3 File Input/Output

To read from or write to a file, you must declare a variable (an object) of type ifstream or ofstream, then connect it to a file with open(). Definitions for both stream types are accessed with #include <fstream.h>:

```cpp
#include <fstream.h>

ifstream inp_file;
ofstream out_file;

inp_file.open( "data.txt", ios::in );
out_file.open( "result.txt", ios::out );

if ( !inp_file ) {
    cerr << "Failed to open input file "data.txt"\n";
    return;
} else if ( !out_file ) {
    cerr << "Failed to open output file "result.txt"\n";
    return;
}

while( inp_file >> i ) {
    out_file << i << " ";
}
out_file << "\n";
```
The above code fragment opens an input file named data.txt, then reads a sequence of characters representing integer values (i.e., groups of characters that form valid integer values like “123” or “5”, with each group separated by one or more spaces or newlines). Each value is written to an output file named result.txt.

Methods used by cin can be used by any ifstream object, and methods used by cout can be used by any ofstream object. This is why we were able to use redirection operators in the above example to read and write integers.

C++ provides a number of additional methods specifically designed to support file I/O through ifstream and ofstream objects:

- open( char *name, openmode mode )
  Opens the file name for reading or writing. For ifstream objects, mode must be ios::in (read). For ofstream objects, mode can be either ios::out (write) or ios::app (append). An already-existing file opened for ios::out is automatically emptied (i.e., its contents are deleted and its size is reset to zero bytes).

- close()
  Closes any previously opened file stream.

- !
  Returns true if the stream object opened successfully, false otherwise.

- seekg( off_type offset, ios::seekdir dir )
  Seeks within an ifstream object, moving offset bytes from starting position dir. Valid positions are ios::beg to seek from the beginning of the file, ios::end to seek from the end of the file, and ios::cur to seek from the current position in the file. Note that for ios::beg, offset must be positive or zero, while for ios::end, offset must be negative or zero. For ios::cur, offset may be positive or negative, depending on where the file position is currently located. In all cases, you must ensure that you do not attempt to seek past the beginning or the end of the file.

- seekp( off_type offset, ios::seekdir dir )
  Seeks within an ofstream object. Arguments are identical to seekg().

It is also possible to open a file for simultaneous input and output with a fstream object. fstream is a subclass of iostream, and can therefore use all the methods that ifstream and ofstream objects support. To open for both reading and writing, use a bitwise OR to specify multiple modes on open:

```c++
#include <fstream.h>

int i;
fstream io_file;

io_file.open( "io_file.txt", ios::in | ios::out );
io_file >> i;
io_file.seekp( 0, ios::end );
io_file << i;
```

5 ifstream is a subclass of istream, and ofstream is a subclass of ostream, so they inherit the ability to act like istream and ostream objects, respectively.
9 COMPILING AND LINKING

A compiler converts source code into machine language instructions that can be executed on the machines you are using. gcc and g++ on Unix are the standard C and C++ compilers. Microsoft DevStudio on Windows includes access to Microsoft’s C and C++ compilers. Below are a number of terms you should become familiar with.

- **source file**: program code entered with a text editor, in our case, the code is C++ code; these files are **compiled** and **linked** to produce executable files;

- **object file**: a source code file which has been compiled but not linked; an object file contains a compiled (machine language) version of the source code, and a symbol table which indicates where various functions and variables are located in the object file;

- **executable file**: a machine language file which can be executed on the machines you are using; a source code file is not an executable file; on the other hand, the file you get after compiling and linking your source code is an executable file;

- **compile**: the conversion of a source file into a machine language file (and symbol table); at this point, the machine language file is an object file, NOT an executable file; the file must be **linked** to produce an executable file, and

- **link**: bundles one or more object files together to form a single executable file; linking ensures the various object files can communicate with one another, if necessary.

One or more C++ source code files are converted into an executable file by successfully compiling and linking them.

**Note**: From this point, I will talk specifically about the Unix commands needed to compile, link, and run a program. Equivalent functionality exists in DevStudio via the Compile, Build, and Start commands.

We convert a C++ source code file into an executable file (i.e., we can compile and link the file) using the g++ command. To compile and link the program foo.c we would type the following at the command line:

```
g++ foo.c
```

This creates an executable file with a default name a.out. We execute the program by typing its name at the command prompt. If we wanted to call our executable file some specific name, like foo, we would type:

```
g++ foo.c -o foo
```

The -o option tells g++ to name our executable file. If we wanted to compile foo.c without linking, we would type:

```
g++ -c foo.c
```

The -c option tells g++ to compile without linking. Compiling creates an explicit object file called foo.o. To link the object file to produce an executable file, we would type:

```
g++ foo.o
```

---

6 By default, DevStudio decides which compiler to invoke based on a source code file’s extension. If the extension is .c, the C compiler is invoked. If the extension is .cpp, the C++ compiler is invoked.
g++ foo.o -o foo

This produces the executable file foo. The compiler is smart enough to know if an input file is a source code file or an object file. Therefore, the compiler does not attempt to recompile foo.o. It knows the only step required is a link step. Notice that the two steps we just performed are the equivalent of typing `g++ foo.c -o foo`.

There are four types of errors you should expect to encounter while you develop your software:

1. **Syntax**, caused by incorrect syntax in your source code. The compiler will report the file and line where the error occurred.
2. **Link**, usually caused by missing or misspelled procedure or function names. The compiler will report that a certain procedure is undefined.
3. **Runtime**, your program compiles and links, but when you try to run it, the operating system reports an error. There is incorrect logic in your source code.
4. **Invalid**, invalid results were returned by the program, even though it ran without errors.

The methods used to fix each type of error differ. For syntax errors, you can normally just edit the file and fix the offending line. For link errors, you should check your source code to ensure the unknown procedure actually exists, and that you have used the exactly same spelling in all cases. Also, make sure you used `#include` to include the header file (`.h` file) that contains the function's prototype, if necessary. A runtime error requires you to determine where the error occurred, and then carefully scan your source code to see what caused it. You can try using `cerr` statements to track down where the error is happening. You can also use the debug program `gdb` to try to find where the error occurred. Finally, invalid results also require you to try to determine where your program logic is incorrect by carefully scanning your source code. As with runtime errors, you can use `cerr` statements or `gdb` to track down the logic error.

### 9.1 Multi-Module Programs

As we have seen before, there is no rule which says the source code for an executable program needs to be located in a single source code file. Usually, the code for a program is spread throughout a number of files. These files are compiled separately to create object files. The object files are then linked together, to produce the executable program. Remember that by default the compiler performs two steps to create an executable program. Consider the following example:

```
g++ date.cpp time.cpp main.cpp -o date
```

In this example, the compiler compiles three source code files `date.cpp`, `time.cpp`, and `main.cpp`. Each compilation pass produces a temporary object file. After compiling is complete, the three temporary object files are linked together to produce an executable file `date`. The object files are then deleted.

Recall that we can split the compile and link steps, and tell the compiler to produce explicit object files with the `-c` flag:

```
g++ -c date.cpp time.cpp main.cpp

g++ date.o time.o main.o -o date
```

This produces the same result as before (an executable program called `date`), however, it also creates three object files `date.o`, `time.o`, and `main.o`, which are not deleted by the compiler after the link step.
Why would we care about creating separate object files? If your program is made up of multiple source code files, this means you will only have to recompile the files that you actually modify to recreate your program. For example, supposed we edit and change `time.cpp`. We can rebuild `date` as follows:

\[
\begin{align*}
g++ & -c \text{ time.cpp} \\
g++ & \text{ date.o time.o main.o -o date}
\end{align*}
\]

That is, we only recompile the source code that has actually changed to produce a new, updated object file. For code that hasn’t changed, the previous object files (`date.o` and `main.o`) already exist.

If we didn’t create separate object files, we would have to recompile all the source code in our program every time we wanted to create an updated executable. This can be time consuming, particularly if the program has even a moderate number of source code files.

You should try to keep your source code files to a manageable size, grouping procedures logically in separate files. This will lead to easier program development and maintenance.

### 9.2 Debugging

One file compiler flag you may want to use is `-g`. This is specified during the compile phase, for example:

\[
\begin{align*}
g++ & -g \text{ date.cpp time.cpp main.cpp -o date} \\
g++ & \text{ -c -g date.cpp time.cpp main.cpp} \\
g++ & \text{ date.o time.o main.o -o date}
\end{align*}
\]

- `g` tells the compiler to embed debug information in the object files (and thus in the resulting executable file). If your program produces errors, the `gdb` debugger can then be used to try to locate and correct the errors. It is often much easier to find errors using the debugger, rather than trying to search the code manually, or to use print statements to isolate errors.

`gdb` is invoked on an executable file from the command line as `gdb executable_file`, for example:

\[
gdb \text{ date}
\]

There are numerous online manuals and tutorials for `gdb`. See these resources for more detailed information on how to use the debugger.