# C++ Tutorial

# Part I : Procedural Programming

C. David Sherrill
School of Chemistry and Biochemistry
School of Computational Science and Engineering
Georgia Institute of Technology

## Purpose

- To provide rapid training in elements of C++ syntax, C++ procedural programming, and C++ object-oriented programming for those with some basic prior programming experience
- To provide a handy programming reference for selected topics
- To provide numerous, actual C++ code examples for instruction and reference

## Why C++?

- “Intermediate”-level language: allows for fine (low-level) control over hardware, yet also allows certain complex tasks to be done with relatively little code (high-level)
- Good for scientific applications: produces efficient, compiled code, yet has features that help one develop and maintain a complicated, large code (e.g., namespaces, object-oriented design)

## Recommended reading

- These notes were developed during my reading of "Sams Teach Yourself C++ in One Hour a Day," 7th Edition, by Siddhartha Rao (Sams, Indianapolis, 2012). I recommend the book, it's readable and to the point.
- A good mastery of C++ will probably require working through a book like that one, and doing some examples; notes like these only serve as a basic introduction or a quick review

## A Note on C++11

- This was originally supposed to be C++0x, with the "x" filled in according to the year the new C++ standard was finalized (e.g., C++09 for 2009). However, the standard took longer than expected, and was only formalized in 2011. So, C++11 is what was formerly referred to as C++0x.
- As of 2013, the new C++11 standards are not yet fully implemented in many compilers.  However, I will try to note when any part of this tutorial is relying on new C++11 syntax.

# Chapter 1: Real Basics

- Very basic structure of a program
- Editing and compiling a program
- Basic printing with cout
- Basic input with cin
- A little about variables

## A simple program: hello, world

```
// The next line includes the file iostream, which defines cout below
#include <iostream>

// Every C++ program should have a function called main that returns
// an integer
int main()
{
  // Write "hello, world" to the screen
  std::cout << "Hello, world!" << std::endl;

  // Return 0 to the OS, indicating success
  return(0);
}
```

## Dissecting the example program

- Lines that start with // are comment lines and are ignored by the C++ compiler. Comments in more complicated programs are very important to help you remember what you did and why.
- Comments can, alternatively, be begin with /* and ended with */ (and can, in this form, span multiple lines)
- #include <iostream> is called an “include statement” and inputs the file iostream at the top of the file; this “header file” contains definitions (like std::cout) that we use later in the program
- Every line that actually does something (print, return) is called a “statement” and needs to end with a semi-colon. “Include statements” (see above) or the first line of a function definition (e.g., int main()) do not need to end in semi-colons.
- Every C++ program must have a function called “main” and it should return an integer. A return value of 0 is usually used for success. Everything within the curly braces {} is part of the main() function.

# Very simple printing

- The line
  std::cout << “Hello, world!” << std::endl;
  actually does the printing.  The “<<” operator pushes things onto the “output stream”, std::cout.  The “std::” prefix just means that the “cout” object lives in the “std::” (pronounced: standard) namespace. Namespaces give us a way to specify which “cout” we're talking about, in case there were more than one.
- The “std::endl” is just code for an “end of line” character; directing this to std::cout will cause a line break in the printing to the screen.

## Continuing statements across lines

Typically, a C++ compiler does not see “white space” (tabs, extra spaces, line feeds, carriage returns, etc.). Hence, it's ok to break up a statement across multiple lines, such as this:

```
std::cout << "Hello, world!"
  << std::endl;
```

One exception to this is that it's not ok to put a line break in the middle of a string to be printed. To do that, you need to use the “line continuation character,” \, like this:

```
std::cout << “Hello, \
  world!\n” << std::endl;
```

Note that statements continuing into a new line are usually indented. This is not required but is standard practice and makes the code more readable.

## Typing the program

- To test the program, you need to type it in.  Programs need to be typed into “plain text” files (like those created by the Windows program Notepad, or the Linux programs vi, emacs, gedit, etc.).  You can't use a standard word processor, because word processors do not create “plain text” files.
- You must be very careful to type in everything exactly as required by C++ syntax (semicolons where they need to be, etc.)
- Files containing C++ code should end with a suffix like “.cc” or “.cpp”.  You can find the code for all the examples in these notes in the files accompanying the notes, under the relevant chapter.  This one is under “ch1/hello.cc”.

## Compiling and running

- Now that we have a program, we can compile and run it.  It's supposed to print the line “Hello, world!” to the screen, and that's it.
- In Linux, we can compile hello.cc by going into the directory with that file, and typing
  g++ -o hello hello.cc
  assuming we're using the GNU C++ compiler, g++. This probably also exists as “c++”.  The part “-o hello” tells the compiler to output the program to a file called “hello.”  This is the program we will actually run.
- To run the program, from the directory containing it, just type “./hello”.  The “./” indicates that the file is in the current working directory.  Running the program should create the message, “Hello, world!” on the screen!
- I'm not including notes on compiling and running under Windows. There should be examples of this process somewhere on the internet.

## More about namespaces

- We mentioned above that the “std::” prefix in front of “cout” and “endl” denotes that these are items from the “standard namespace.” This provides a way to denote which “cout” or “endl” is meant, in case these symbols are used elsewhere in the program in a different context.
- On the other hand, it is tedious to keep typing these “std::” prefixes, especially for items we may use frequently. If we avoid naming any other things “cout” and “endl”, we can tell the compiler to assume a “std” prefix:

```
int main()
{
 using namespace std;
 cout << “Hello, world!” << endl;
 return 0;
}
```

## More about namespaces

- In the previous example, if a symbol isn't known, the compiler will try appending a “std::” in front of it, and then search again. Maybe this is a bit overkill if we're only using “cout” and “endl” out of the std namespace. Alternatively, we can specifically point out that it's only these two symbols we want to avoid typing “std::” in front of. We can replace the line

  ```
  using namespace std;
  ```

  with the lines

  ```
  using std::cout;
  using std::endl;
  ```

## More Detailed Printing

- This cout printing is the fancy new C++ way. Sometimes we want a little more control over the formatting of the printing, which can be more directly achieved using older, C-style printing. And the syntax looks more obvious.
- C++ style:
  cout << “Hello, world!” << endl;
- C style:
  printf(“Hello, world\n”);
- printf() is a C function that prints things to the screen according to some specified format (no special format needed for this example). The “\n” character denotes a line break (does the same thing as endl).

## Hello, world! With C-style printing

```
// Don't need iostream anymore since we're not using cout
// but we do need to include stdio.h to make printf()
   available
#include <stdio.h>

int main()
{
  // Write "hello, world" to the screen
  printf("Hello, world!\n");
  return(0);
}
```

# More printing

Example printing.cc (part 1 of 2):

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

// Declare a function for some cout printing
void print_cout();
// Declare a function for some printf printing
void print_printf();

int main()
{
  // First do cout printing
  print_cout();
  // Now do printf printing
  print_printf();

  return(0);
}
```

# More printing, cont'd

Example printing.cc (part 2 of 2):

```
void print_cout()
{
  cout << "My name is " << "David" << endl;
  cout << "Two times two is " << 2*2 << endl;
  cout << "2 / 3 = " << 2/3 << endl;
  cout << "2.0 / 3.0 = " << 2.0/3.0 << endl;
}

void print_printf()
{
  printf("My name is %s\n", "David");
  printf("Two times two is %d\n", 2*2);
  printf("2/3 = %d\n", 2/3);
  printf("2.0/3.0 = %.2f\n", 2.0/3.0);
}
```

# First look at functions

- The example program above does a few things: (1) introduces us to the use of functions in a program (besides main()), (2) provides some more examples of printing (strings and results of arithmetic), (3) compares and contrasts cout vs printf() style printing, (4) shows examples of providing formats to printf() printing
- A function *declaration* looks like this and must occur before the function is called:
  void print_cout();
  and it declares that print_cout() is a function, and it doesn't return anything (return type void). It also doesn't take any arguments (if it did, they would be listed between the parentheses)
- A function *definition* can occur anywhere and provides what the function actually does, e.g.,
  void print_cout()
  {
  … function goes in lines here …
  }
- There are no “return” statements in these functions because they don't return anything (that's why their return type is listed as “void”)

## Printing examples

- This line shows how multiple strings can be concatenated and send to cout:
  cout << "My name is " << "David" << endl;
- This line shows how to print an arithmetic result using cout:
  cout << "Two times two is " << 2*2 << endl;
- The same thing with printf uses a “format string.” The “%d” symbol means “put an integer here, and the integer will come after the end of the format string”:
  printf("Two times two is %d\n", 2*2);
- “%s” means printf should insert a string here:
  printf("My name is %s\n", "David");

## Integer vs floating point arithmetic

- Notice that the line
  cout << "2 / 3 = " << 2/3 << endl;
  (or it's printf() equivalent) prints the number 0. That's because 2 and 3 are interpreted as integers, and the result will also be computed as an integer (rounded down), which is 0. If we want a floating-point result, we need to do this:
  cout << "2.0 / 3.0 = " << 2.0/3.0 << endl;
- The above line prints out “0.666667”. What if we wanted more (or fewer) digits? We can give the number of digits with printf(). The following line tells printf() to expect a floating point number after the format string, and to print it to two digits after the decimal:
  printf("2.0/3.0 = %.2f\n", 2.0/3.0);

## Example input

- Finally, we conclude this short introduction with an example of how to read data from the terminal.  We wouldn't normally do it this way (you'd typically take command-line arguments or else read from a data file).  But if you ever want to prompt the user for interactive input, the next example shows how you could do it.

## Using cin

Example cin-example.cc:

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

int main()
{
  // Declare an integer to store the user's input number
  int number;
  // Declare a string to store the user's input name
  string name;
  cout << "Enter an integer: ";
  cin >> number;
  cout << "Enter your name: ";
  cin >> name;

  // print the result
  cout << name << " entered the number " << number << endl;
  return 0;
}
```

## Using cin

- If you compile and try this example, you'll see the program prints
Enter an integer:
and then waits for the user to type a number and hit Enter.  It will then prompt for the user's name and wait for the user to type it and hit enter.  Then the program will print something like this:
David entered the number 4
- You can see that “cin” is for input just like “cout” is for output.  But “cin” must have a *variable* to store the information in.  That's why we needed to declare two variables, one an integer (for the number), and one a string (for the name)

## Multiple statements on one line

As a side-note, this is a good time to point out that a line can contain more than one statement. For example, in the previous program, the two lines

```
    cout << "Enter an integer: ";
  cin >> number;
```

could just as easily be written as

```
   cout << "Enter an integer: ";  cin >> number;
```

## A little about variables

- Just like functions, we must *declare* variables before we can use them. This tells the program “I'm going to use a variable with this name, and it will have this type.” The *type* of the variable might be an integer, a floating-point number, a string, etc.
- Variables are *defined* whenever we give them a value (the value can change during the program, that's why it's called a variable). Here, we give them a value by pushing into them whatever the user input, with a line like “cin >> number”.
- Strings were not originally built in as a basic variable type to C or C++. That's why we need to add the “#include <string>” line at the top, to set up the program to use strings. Apart from this minor annoyance, C++11 has significant support for strings.

## Summary

- This chapter has provided a very brief introduction to some basic concepts like typing in and compiling a program, printing, accepting user input, functions, and variables.  We will examine these concepts in more detail in the following chapters.

# Chapter 2: Variables and Constants

- Declaring and defining variables and constants
- Variable types
- How to change the value of a variable
- Scope of a variable
- Size of a variable in memory
- Inferring data type using “auto”
- Using typedef as a shorthand for long names of variable types
- Constant expressions (constexpr)
- Enumerated data types (enum's)

## Variables and Constants

- A *variable* is a quantity that can change during the course of a program
- A *constant* is a quantity that does not change during the course of a program
- A variable would be useful for storing a countdown timer, for example. A constant would be useful for holding the value of π to some desired accuracy (like 3.1415926)
- Both variables and constants are held in some memory location by the program. There are also different *types* for variables and constants (integers, floating-point numbers, strings, etc.)
- Other than not being able to change, constants behave like variables: they are both quantities stored in some memory location, and having a type. We will often use "variable" in a generic sense to mean a variable or a constant. Admittedly, this may be confusing, but this is often how programmers think of them (and they may even speak of a "constant variable"!)

## Declaring Variables

- To *declare* a variable, you give the type of the variable and the name of the variable. For example,

```
int x;
double y;
bool z;
```

where x is an integer, y is a double-precision floating point number, and z is a boolean (true or false)

- Once a variable has been declared, the compiler knows its name and its type, and it can be used later in the program

## Defining Variables

- To *define* a variable, you simply give it a value. You must have previously declared the variable. For example,

  ```
  int x;
  x=10;
  ```

- It is more common to combine variable declaration and definition in a combined statement like this:

  ```
  int x = 10;
  ```

## Variable Names

- Can't start with a number
- Can't contain spaces
- Can't contain arithmetic operators (+, -, *, / denote addition, subtraction, multiplication, and division)
- Can't be the same as a C++ keyword

## Example: Using and Changing Variables

Example vars1.cc:

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

int main()
{
  int number;

  cout << "Enter an integer: ";
  cin >> number;

  number = number * 2;
  cout << "Your number * 2 = " << number << endl;
  return 0;
}
```

# Changing a Variable

- From the previous example, we see that we can change the contents of a variable with a statement like this:
  number = number * 2
  in this case, the old value of “number” is multiplied by 2, and then the result is stored back in the variable “number”. Effectively, “number” has its value doubled.
- This destroys the previous value of the variable.  If we wanted to keep the old value of the variable *and* have the new value available, we'd need two variables, and could do something like this:
  int new_number = number * 2

## Scope of a Variable

- A variable's “scope” consists of the parts of the program where the variable is known
- In general, a variable is only available in the function where it is declared (this is known as a “local variable”); you can't use it elsewhere in another function without taking extra steps
- A variable that is made available within an entire file or to an entire program is called a “global variable”
- You can have two local variables in two different functions with the same name. They won't have anything to do with one another. Changing one will not change the other.

## Scope Example

scope.cc example:

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

void print_five();
int main()
{
  int number = 4;
  print_five();
  cout << "number = " << number << endl;
  return 0;
}
void print_five(void)
{
  int number = 5;
  cout << "number = " << number << endl;
}
```

## Scope Example

- In this example, we demonstrate that the two variables named “number” have nothing to do with one another. One is local to the function main(), and the other is local to the function print_five()
- Note we had to declare the function print_five() before we were able to use it. We could do that inside main() (making the function available for use within main()), or up above, outside main, making the function available to any functions in scope.cc. We chose the latter. This makes the function “global” within this file. Function declarations are thus *also* local or global, just like variable declarations!
- The variable “number” is set to 4 in main(), and it stays 4 inside main(), even though another variable with the same name is set to 5 in the function print_five(), which is called after “number” in main is set to 4, but before it is printed in main()

## Global Variables

- Suppose we didn't want this behavior.  Instead, we want a variable, “number”, to be available as a single variable throughout our file.  We can take a clue from how we made the function print_five() in our previous example visible to the entire file (and not just within main) by placing it up top in the file, outside any function, e.g., we can declare a variable like this:

```
int number = 4;
int main()
{
…
}
```

## Global Variable Example

globals.cc example:

```
#include <iostream>
using namespace std;
void print_five();
int number = 4;

int main()
{
  print_five();
  cout << "number = " << number << endl;
  return 0;
}

void print_five(void)
{
  number = 5;
  cout << "number = " << number << endl;
}
```

## Global Example Comments

- In the previous example, “number” became a *global variable* whose scope extends to the entire file. There is no longer a need to declare the variable in each function, it is declared once at the top of the file.
- Any change to the variable anywhere in the file is now reflected throughout the entire file

## Types of Variables

- bool : true or false
- int: an integer
- float: a floating point number (like 2.385, etc.)
- double: a double-precision floating point number (like above, but can carry more digits). This is preferred over float for scientific applications to help avoid roundoff errors.
- char : a single character (e.g., 'a', 'b', etc.)

# Special Versions of the Common Variable Types

- There are different *modifiers* that can be applied to some of the standard variable types. For example:
- unsigned int : an integer that can't be negative
- int : on most machines, this only allows values in the range -2,147,483,648 to +2,147,483,647 (about +/- 2E9)
- short int : on most machines, uses less memory and allows values in the range -32,768 to +32,767
- long int : allows larger integers than a regular int; on most machines, up to +/- 9E18
- long long int : may allow even larger numbers than a long int (on many machines, it's really just the same as a long int)

## Special Versions, cont'd

- Can combine “unsigned” with “short,” “long,” or “long long” to get, for example, short unsigned int
- Notice that “unsigned” allows us to store numbers up to twice as large (at the expense of not having a sign). This is because the one bit (0 or 1) formerly used for the sign bit is now available, and each extra bit allows us to count up to a number about twice as large as before the bit was added. (With n bits, can count up to $2^n$-1).

# Numbers of Bits

- On a modern 64-bit machine, most quantities are processed 64-bits at a time (one “word”). For example a “double precision” floating point number on a 64-bit machine is 64-bits (and a “float” is half this, or 32 bits)
- 8 bits per “byte”
- How many bits in an integer, float, double, etc, are dependent on the machine and the compiler. But we can use the “sizeof()” command to get the system to tell us how big each variable type is.

# wordlength.cc

Example wordlength.cc:

```
#include <stdio.h>
int main()
{
  printf("Size of char               = %ld\n", sizeof(char));
  printf("Size of float              = %ld\n", sizeof(float));
  printf("Size of double             = %ld\n", sizeof(double));
  printf("Size of long double   = %ld\n", sizeof(long double));
  printf("Size of short int          = %ld\n", sizeof(short int));
  printf("Size of int                = %ld\n", sizeof(int));
  printf("Size of long int           = %ld\n", sizeof(long int));
  printf("Size of long long int = %ld\n", sizeof(long long int));
  return 0;
}
```

## Comments on wordlength.cc

- The sizeof() command takes one argument, the data type (placed in parentheses), and it returns how much memory is used to store that data type, in bytes
- In the previous example, we use C-style printing. The format string now contains “%ld” because in C++, the sizeof() function returns an (unsigned) long integer
- Try this on your system. On my system with the GNU G++ compiler and a 64-bit machine, I get the following results: char = 1, float = 4, double = 8, long double = 16, short int = 2, int = 4, long int = 8, long long int = 8. It's interesting that even on a 64-bit machine, by default only 32 bits are used to store an integer, and it takes “long int” to force the compiler to use 64-bits.

# overflow.cc

A simple example demonstrates why it's important to use a data type big enough to hold the required data. If we want to multiply 2 billion by 2, this will overflow a regular integer (causing the result to “wrap around” to a negative number). But it works when using long int's

```
#include <stdio.h>
int main()
{
  int p = 2000000000;
  int q = p * 2;
  printf("p*2 = %d\n", q);

  long int lp = 2000000000;
  long int lq = lp * 2;
  printf("p*2 = %ld\n", lq);
  return 0;
}
```

Output:

```
p*2 = -294967296
p*2 = 4000000000
```

# bool and char types

- A variable of type bool can be true or false:
  bool found = true;
- A variable of type char holds a single character. Internally, the character is actually represented as an integer, using the ASCII codes that map a character to an integer.  Thus, a char can be processed either as a character or as an integer (although the integer can only go up to 255 since it's just one byte)
  char s = 'e'; // store the letter 'e'

## Using auto to Infer Type

- In what at first appears to be a surprising, laziness-enabling feature, C++11 can try to infer the data type of a variable based on the value that is used in the definition of the variable, e.g.,
  auto Found=false; // can deduce it's a bool
  auto Number=20000000000; // use a long int
- This seems somewhat pointless because the programmer really ought to know the datatypes. But it can be useful as a way to avoid figuring out more complicated data types later on, when we start using advanced features (e.g., what's the data type of an iterator over a standard vector of integers? A vector<int>::const_iterator. Maybe easier to let the compiler figure that one out...)

## Using Typedef

- Sometimes, the datatypes can have rather long names (e.g., unsigned long int). These can be tedious to type. We can use “typedef” to create a shorthand notation in such cases:

```
typedef unsigned long int BIGINT;
BIGINT veclen;
BIGINT offset;
```

- If you have several variables of this type, then the code is easier to type and read using typedef's.

## Constants

- Constants have a type and a value and take up memory, just like variables. But their values are not supposed to change during the program.
- Advantages: By declaring something as a constant, you're telling the compiler to watch out and not allow the value to change. You are also providing the compiler some extra information it might be able to use to speed up the code.
- Disadvantages: Once you start using constants, you can't treat them later on in the program as regular variables. This can be annoying if you try to use functions later on that expected real variables, not constants. Keeping everything consistent is the price you pay for declaring a constant.

## Literal Constants

- We have already encountered a sort of trivial case of constants, e.g.,
  int number=4;
  where the “number” is a variable, but the right-hand side (4) is certainly a constant.
- Another example would be a string constant, like “Hello, world!” in
  cout << “Hello, world!”

## Declaring a constant

- A “const” specifier is placed before a normal variable declaration. For example,
  const double pi = 3.1415926;
  says that “pi” is a double-precision value that will not change during the program (certainly we won't be re-defining pi during the course of the program!)
- After something is declared a constant and defined, its value cannot be overwritten (e.g., can't say pi = 5.0 at some future point in the program)
- Otherwise, we use this just like we use a regular variable

## constexpr

- constexpr is similar to “const”, but it means “constant expression.” It is new to C++11. It can be used to indicate that the results of a function are always the same (constant) and can be evaluated once at compile time, and not every time the function is called. This can speed up the program if the function is called often.

  ```
  constexpr double EstPi() { return 333.0 / 106.0; }
  constexpr double TwoPi() { return 2.0 * EstPi();}
  ```

## Enumerated Constants

- C++ also has an elegant “enumerated” datatype that lists a set of options symbolically. Internally, the compiler converts each option into an integer. The user can specify what integer to map to what enum option --- but working with the internal integer representation goes counter to the spirit of enums, where the whole point is that using a word symbol is more natural than a number
- First one specifies an enum datatype by giving it a name and listing all the possible values it can have. Then this datatype can be used to create new variables. The variables have to have a value that is one of the allowed values for the enum.

## Enum Example

```
enum Directions {
    South,
    North,
    East,
    West
   };
   Directions heading = South;
```

## #define

- With support for constants in C++, there is no longer a reason to use the pre-processor directive #define. However, this was the way constants were specified in C, and hence #define statements are still frequently encountered.
- #define PI 3.1415926
  would define the symbol PI with the value of 3.1415926. It acts as an alias. Everywhere PI is encountered in the code, it is replaced by “3.1415926”. This is inferior to the new mechanism, because the compiler doesn't know anything about the datatype of PI in the #define version.

## Naming Variables

- The programmer should take extra care to use meaningful variable (or constant) names. This makes the code easier to understand, which helps everyone (including the programmer, if he or she ever has to work on the code again! You'd be surprised how much of your own code you can forget after a year or two.)
- For example, variable names like found, converged, or TotalEnergy are superior to f, con, or E.

# Chapter 3: Arrays and Strings

- Static arrays
- Dynamic arrays
- C-style strings
- C++-style strings

## Arrays

- An array is an ordered collection of items of the same type
- In C++, an array is accessed by giving the name of the array and which element of the array you want (e.g., value[4])
- Array numbering in C++ starts from 0.  So, the *first* element in an array is indicated by [0], the *second* element by [1], etc.

## Declaring and defining static arrays

- A *static* array is one whose length is defined once (as a constant) and does not change during the program (the length of the array just refers to the number of elements contained in the array)
- To create an array called “coord” to contain 3 values (for x, y, and z), you could do this:
  double coord[3];
- To set the values, you could use
  coord[0] = 0.1; coord[1] = 3.4; coord[2] = 9.7;
- Alternatively, you could set the values at the same time as declaring the array:
  double coord[3] = {0.1, 3.4, 9.7}; // contains initializing values
- You can access array elements using variables as well as constants, e.g.,
  next_coord = coord[i]; // i is an integer (from 0-2)

## More about initialization

- To initialize every element of the array to the same value, just do something like this:
double coord[3] = {0.0}; // all three values will be 0.0
- You can also let the compiler figure out the length of the array if you give all the initial values, like this:
double coord [] = {0.1, 0.2, 0.5}; // compiler knows a "3" should go in the []

## Arrays in memory

- You can have an array of any type of data (double-precision numbers, integers, characters, even user-defined datatypes)
- When a static array is declared, the compiler figures out how long it is, and how much memory it takes to store one of the items of data. Then it creates a contiguous stretch of memory whose size is equal to the memory required per item times the number of items
- For an array “coord” of 3 double-precision words, coord[0] is stored first in a space consisting of sizeof(double) bytes, followed by coord[1] taking another sizeof(double) bytes, followed by coord[2] taking a final sizeof(double) bytes. It's up to the compiler to worry about exactly where in memory all this is located (the symbol “coord” will “point” to the beginning of the allocated memory)

## Using arrays

- Get the n-th element of an array:
  result = value[n];
- Store a number in the n-th element of an array:
  value[n] = result;
- Do NOT attempt to get or set an array element beyond the length of the array. In our coordinate example, trying
  result = coord[3]; // fails!
  is a bad idea because “coord” only has length 3 (and therefore only elements [0], [1], and [2] --- remember, we start counting from 0)
- Setting an array element beyond the allocated length can cause “segmentation fault” runtime errors. If you're unlucky, doing this might accidentally overwrite some other chunk of memory used elsewhere in the program, leading to a bug that is very hard to track down. This is one of the most common C++ programming errors. Tools like “valgrind” exist to help you catch such errors.

# Arrays in memory

- You can have an array of any type of data (double-precision numbers, integers, characters, even user-defined datatypes)
- When a static array is declared, the compiler figures out how long it is, and how much memory it takes to store one of the items of data. Then it creates a contiguous stretch of memory whose size is equal to the memory required per item times the number of items
- For an array "coord" of 3 double-precision words, coord[0] is stored first in a space consisting of sizeof(double) bytes, followed by coord[1] taking another sizeof(double) bytes, followed by coord[2] taking a final sizeof(double) bytes. It's up to the compiler to worry about exactly where in memory all this is located (the symbol "coord" will "point" to the beginning of the allocated memory)

## Multi-Dimensional arrays

- Suppose instead of one set of coordinates in 3D (x, y, and z values), we need multiple sets of 3D coordinates. We need an array of arrays. We can do this in C++.
- Let's say we have two sets of 3D coordinates. They could be initialized as follows:
  double coordinates[2][3] = { {0.1, -0.3, 4.2}, {1.0, 0.1, 0.2} };
- Now coordinates[0] refers to the first set of coordinates, and coordinates[1] refers to the second set of coordinates. To get the x coordinate of the second set of coordinates, that would be coordinates[1][0] (assuming we stored x first, then y, then z). The z coordinate from the first set of coordinates would be coordinates[0][2].
- Clearly 2D arrays like this are logical ways to store 2D matrices. The first index refers to the row of the matrix, and the second index refers to the column, e.g., matrix[row][column].

## Dynamic arrays

- Remember that a *static array* is one whose dimensions are given as constants in the array declaration, and they do not change during the program (although the *contents* of the array can certainly change!)
- A *dynamic array* is one in which the dimensions are variables determined at runtime; the size of the array might change during the computation
- There are various ways to create dynamic arrays in C++.  One of the easiest and more modern ways to do this is using the built-in “vector” capability

## Dynamic arrays with std::vector

vector.cc:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> values(2);

  values[0] = 31;
  values[1] = 42;

  cout << "Length of array 'values' is " << values.size() << endl;
  cout << "values[0] = " << values[0] << endl;
  cout << "values[1] = " << values[1] << endl;

  // Now make the array grow by another value!
  values.push_back(53);
  cout << "Length of array 'values' is " << values.size() << endl;
  cout << "values[2] = " << values[2] << endl;

  return 0;
}
```

Output:

```
Length of array 'values' is
2
values[0] = 31
values[1] = 42
Length of array 'values' is
3
values[2] = 53
```

## Comments on vector.cc

- There are some slightly mysterious things going on in the previous example: the array declaration takes the type inside < > symbols, there is a “push_back” function, etc.  Don't worry, we'll get to these things later.
- Nevertheless, it should be basically clear from the context what the program does and how it works.

## Strings

- In C, there was not a built-in data type for strings. Instead, a string was treated as an array of characters. Knowing this is useful because much C or early C++ code exists that uses such kinds of strings, and it is good to be able to interact with this code.
- Newer versions of C++ include a built-in string type

## C-style strings

- Allocate just like an array, it's just that it's now an array of characters
- A special “null character” (denoted `\0`) is stored at the end of the string to indicate the end-of-string. If this terminator is missing, then bad things happen (e.g., statements that try to print the string will keep printing characters until they eventually reach this character somewhere else in memory),
- To store a 5-character name, one must allocate 6 characters: 5 for the name, and one for the null character.
- Keeping up with all these details is why C-style strings are disfavored compared to the newer C++-style strings
- Other useful functions for C-style strings are strcpy (copy a string), strlen (get a string length), strcmp (compare two strings --- note, if they match, the return value is zero, which is a little confusing), etc. To use these functions, #include <cstring>

## C-style strings

```
c-string.cc:
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
  char name[] = "David Sherrill";
  cout << name << endl;

  name[5] = '\0';
  cout << name << endl;

  strcpy(name, “Rollin”);
  cout << name << “ has “ < < strlen(name) << “ characters.” << endl;
  return 0;
}
```

Program output:

```
David Sherrill
David
Rollin has 6 characters
```

## C++-style strings

- Safer for programming because they can scale their size dynamically and the programmer doesn't need to worry about making sure the terminating null character is in the right place
- Easier to manipulate with high-level syntax
- To use, #include <string>

## C++-style strings

c++-string.cc:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string FirstName("David");
  string SecondName("Sherrill");

  string FullName = FirstName + " " + SecondName;

  // The next line prints "David Sherrill" with a space between names
  cout << FullName << endl;

  return 0;
}
```

## C++-style strings

- Safer for programming because they can scale their size dynamically and the programmer doesn't need to worry about making sure the terminating null character is in the right place
- Easier to manipulate with high-level syntax
- To use, #include <string>

# Chapter 4: Operators and Expressions

- Math operators
- Logic operators
- L-values and r-values
- Expressions

## Operators

An “operator” is something that transforms data. Common operators in C++ are as follows:

- = : Assignment operator, takes the value on the right and puts it into the variable on the left.
- + : Addition operator
- - : Subtraction operator
- * : Multiplication operator
- / : Division operator
- % : Modulus operator (remainder after a division)
- … and others to be discussed in this section

## Assignment operator

- A typical use of the assignment operator is as follows:
  int x = 3;
  This declares a variable called x, which is an integer, and assigns to it the value 3.
- Note that the object on the left-hand side has to be a variable that can hold a value. This is often called an “lvalue” by the compiler (something on the left-hand side of an assignment operator). Variables make valid lvalues because they can take a value. But constants, for example, are not lvalues because they cannot change their values. So, a statement like
  5 = 3;
  is invalid because 5 is not a valid lvalue
- The above example line may seem like one no programmer would ever write. But this is actually a very common mistake, because programmers often write something like this as part of a test to check if two values are equal. This is NOT to be done with the = operator. To check equality, use the == operator (two equals signs), described later.
- Things that can be on the right-hand side of an assignment operator (variables, constants) are called “r-values”

## Math operators example

mathops.cc:

```
#include <iostream>

using namespace std;

int main()
{

  cout << "2+3 = " << 2+3 << endl;
  cout << "2-3 = " << 2-3 << endl;
  cout << "2*3 = " << 2*3 << endl;
  cout << "2%3 = " << 2%3 << endl;
}
```

Program output:

```
2+3 = 5
2-3 = -1
2*3 = 6
2%3 = 2
```

## Modulus operator, %

- Hopefully everything in the previous example is pretty obvious, but perhaps the last statement might not be:

  ```
  cout << "2%3 = " << 2%3 << endl;
  ```

  prints the value 2.  The modulus operator, %, does a division and then prints the remainder.  So, 2%3 is the remainder of 2 divided by 3.  Remember we're dealing with integers (which is always true if we're using %).  So, 3 does not go into 2 (or, it goes into 2 zero times), so the result of the division is 0, and the remainder is 2.

## Increment and decrement operators, ++ and --

- Very frequently we need to increase the value of a variable by one (especially if we're counting something). We could do this simply by:

  ```
  x = x + 1;
  ```

  but this is so common that there's a shortcut operator for this:

  ```
  x++;
  ```

- Likewise, there is a decrement operator, --, that decrements a variable by 1:

  ```
  x = x – 1;
  ```
  is the same as
  ```
  x--;
  ```

## Increment/decrement example

- incdec.cc:

```
#include <iostream>

using namespace std;

int main()
{

  int x = 3;

  cout << "Started x at " << x << endl;
  x++;
  cout << "After x++, now x is = " << x << endl;
  x--;
  cout << "After x--, now x is = " << x << endl;

}
```

Program output:

```
Started x at 3
After x++, now x is = 4
After x--, now x is = 3
```

## More about ++, --

- We can also include the increment and decrement operators as part a larger statement or mathematical expression.  For example,

  int x =3, y = 5;
  y = x++; // this makes y (and x) be 4
  cout << x++ << endl; // this prints 5

- Note that the above uses of ++ have a dual role: they increment AND they allow the incremented value to be used in an assignment or a print statement (etc.)
- What if you wanted to print the value of x and THEN increment it by one?  This can still be done by using ++ as a *prefix* operator instead of a *postfix operator*, e.g.,
  int x = 3; cout << ++x << endl; // this prints 3 and THEN makes x=4
- Analogous statements hold for the decrement operator --

## Operators +=, -=, *=, /=

- Frequently we want to take a variable, do some math operation on it, and put the result back into the variable. We can do this the long way, e.g.,

  int x = 3; x = x + 2; // yields x=5

  or the short way

  int x = 3; x += 2; // also yields 5, but shorter
- x += 2 is a shortcut for x = x + 2;
- x -= 2 is a shortcut for x = x – 2;
- x *= 2 is a shortcut for x = x * 2;
- x /= 2 is a shortcut for x = x / 2;

## Equality operators (==, !=)

- Earlier we made the point that = is an assignment operator, not a test of equality. Tests of equality use the equality operator, ==. Use of this operator returns a boolean value (true or false). For example:

  ```
  bool n = (2 == 3); // makes n 'false'
  bool m = (1 == 1); // makes m 'true'
  int x = 2, y = 3;
  bool p = (x++ == y); // makes p 'true'
  ```

- != is an “inequality” operator that checks if two quantities are *not* equal:

  ```
  bool n = (2 != 3); // makes n 'true'
  ```

## Relational operators (<, >, <=, >=)

- Just as we can check for equality or inequality, we can also check if values are greater than (>), less than (<), greater than or equal to (>=), or less than or equal to (<=). The results again are booleans.

  ```
  bool n = (2 > 3); // false
  bool m = (1 > 1); // false
  bool p = (1 >= 1); // true
  int x = 2, y = 3;
  bool q = (x <= y); // true
  ```

## The logical NOT operator (!)

- We can reverse the value of a boolean result by the logical NOT operator, denoted by the ! character

  ```
  bool n = (2 > 3); // false
  bool m = !(2 > 3); // true
  ```

- NOT takes 'true' and makes it 'false', or 'false' and makes it 'true'

## The logical AND operator (&&)

- AND (denoted &&) is a “binary” operator, like the + operator, in that it requires two operands to work with. If both operands are true, then the AND results to true. Otherwise, the result is false:

  ```
  bool n = (2 > 3) && (1 == 1); // false
  bool m = (3 > 2) && (1 == 1); // true
  ```

- We can string &&'s together; if we do, ALL the pieces have to be true for the result to be true:

  ```
  bool n = (3 > 2) && (1 == 1) && (2 >= 2); // true
  ```

## The logical OR operator (||)

- OR (||) is also a binary operator.  It evaluates to true if either of its operands are true.

  ```
  bool n = (2 > 3) || (1 == 1); // true b/c (1==1) is
  bool m = (2 > 3) || (3 > 4); // false b/c neither is
  ```

## The logical XOR operator (^)

- The XOR (^), is for “exclusive OR”. It evaluates to true if *one and only one* of the operands is true.

```
bool n1 = (3>2) ^ (1==1); // both true, XOR false
bool n2 = (3<2) ^ (1==1); // one true, XOR true
bool n3 = (3<2) ^ (1==2); // both false, XOR false
```

## Expressions

- An *expression* is simply a bit of code that evaluates to a value.  For example, in the statement
  cout << “2+3 = “ << 2+3 << endl;

2+3 is an expression that evaluates to 5.

- Logic tests are also expressions, because they evaluate to True or False.  For example,

(2+3 == 5)
  evaluates to True

## Use of equality, relational, and logical operators

- The true power of the kinds of operators we've been discussing becomes apparent when we see how they can be used for *program flow control*. That is, we can use them to let the program decide what to do next based on the data it currently has. Although program flow is an upcoming topic, we will go ahead and demonstrate the “if” statement in conjunction with the operators we've been discussing.

## The 'if' statement

- 'if' checks a boolean result and if it is true, then it executes some statement. For example,

```
if (3==2)
  cout << “Wow, 3=2?  I'm surprised” << endl;
```

- If more than one line is to be executed by the 'if' statement, then the lines must be grouped into a “code block” designated by curly braces:

```
if (age > 65) {
  cout << “Qualifies for senior discount” << endl;
  price *= 0.85;  // 15% discount
}
```

- It's a good idea to always use the braces, even if the 'if' needs to execute only one line, because that way if you add a line later, you don't have to remember to add the braces (common mistake that can be hard to spot).
- Lines in the 'if' block to be executed are usually indented to make the program more readable.

## if/else

- Frequently we want to do one thing if the test evaluated by 'if' comes up true, but *another* thing if it comes up false.  We can do that with the 'if/else' combination.

```
if (age > 65) {
  cout << “Qualifies for senior discount” << endl;
  price *= 0.85;  // 15% discount
}
else { // this section executes if (age <= 65)
  cout << “Normal price” << endl;
}
```

## Flow control with more elaborate logic

- The previous tools allow us to contruct rather elaborate flow control for complex situations.
- Suppose we have a senior citizen discount for customers 65 or over.  But we have plenty of customers on Friday, so the discount doesn't apply on Fridays.  But if the customer is 80 or over, the discount will apply any day.
- Analyzing the above rules, we want the discount to apply IF (a) the customer is 80 or over, OR (b) the customer is over 65 and it's not Friday.  The next example shows how to check for this.

# Logic example

## discount.cc

```
#include <iostream>

using namespace std;

int main()
{
  int age = 70;
  string day("Sunday");

  if ((age >= 80) || (age >= 65 && day != "Friday"))
    cout << "Discount applies" << endl;
  else
    cout << "Discount does not apply" << endl;
}
```

Program output:

```
Discount applies
```

## Grouping in logic statements

Notice the use of parentheses in the previous example:

```
    if ((age >= 80) || (age >= 65 && day != "Friday"))
```

The parentheses make it clear that

```
    (age >= 80)
```

is one way to get the discount, and that

```
    (age >= 65 && day != “Friday”)
```

is the other way to get the discount (and since either way works, there is an OR operator between the two possibilities).

If we didn't have the parentheses, it wouldn't be clear that the day != “Friday” condition is grouped with the age>=65 condition. You should always carefully figure out the grouping of logic conditions and apply parentheses where necessary to make the grouping clear to the programmer and the compiler.

## Operator precedence

- Similarly to the previous example, arithmetic operations should be grouped by parentheses to make the order of operations clear to the user
- There are “rules of precedence” that determine what order a complex set of operations will be carried out in.  In principle, these rules could be used by the programmer to write code that will execute correctly.  But it might be impossible to understand without someone doing needless work to trace the order of operations.  Make things easy on yourself and on others.  Just use parentheses.
- For example, the result of the following operation may not be obvious:
  ```
  int x = 10*3-6/2+1;
  ```
  but the following is obvious:
  ```
  int x = (10*3)-(6/2)+1; // 28
  ```
- FYI, * and / are evaluated first, followed by + and -

## Faster && evaluations

Note: In an expression containing a sequence of AND statements, like

if ((x>3) && (y>2) && (z>1))

As soon as any AND conditions fail, the program leaves the expression and moves on to the next statement. Therefore, to speed up execution, put the statement most likely to fail (or easiest to check) first.

# Bitwise operators

- The logic operators (AND, OR, XOR, NOT) each have a corresponding “bitwise” version that acts on individual bits (1's and 0's) making up an integer.
- The bitwise versions use single symbols: & (AND), | (OR), ^ (XOR), and ~ (NOT).
- For example, the integer 4 is represented in binary as 100, and the integer 3 in binary is 011. The binary OR for these two integers is just the binary OR for each bit separately. The first bit yields 1 (1 OR 0 is 1), the second bit yields 1 (0 OR 1 is 1) and the third bit yields 1 again (0 or 1 is 1). So, all three bits were 1, giving an answer of 111, which as an integer is 7. Thus 4|3=100|011=111=7.
- When applied to an integer, >> is a “bit shift” operator and will shift all the bits a given number of places to the right. << shifts the bits a certain number of places to the left. For example, Num >> 2 shifts the bits in integer Num 2 places to the right.

# Bitwise operators example

```
#include <iostream>

using namespace std;

int main()
{
  int z;

  z = 3 & 4; // 011 & 100 = 000 = 0
  cout << "3&4 = " << z << endl;
  z = 3 | 4; // 011 | 100 = 111 = 7
  cout << "3|4 = " << z << endl;
  z = 7 ^ 4; // 111 | 100 = 011 = 3
  cout << "7^4 = " << z << endl;
  z = 4 >> 2; // 100 >> 2 = 001 = 1
  cout << “4>>2 = “ << z << endl;

}
```

Program output:

```
3&4 = 0
3|4 = 7
7^4 = 3
4>>2 = 1
```

## Truth of numbers

In C++, 0 evaluates as false, and all other numbers evaluate as true

## The Ternary ?: Operator

(expression1) ? (expression2) : (expression3)

If expression1 is true, then return the value of expression2, otherwise return the value of expression3

z = (y%2) ? 1 : 0;

In this example, if y is odd, then y%2 has a remainder (making expression1 true), and therefore z=1. Otherwise, z=0.

# Chapter 5: Program Flow Control

- More about if, else, and else if
- switch/case statements
- Loops: while, do, do...while, for
- break and continue statements

# Nested if/else

We can have if/else statements inside if/else statements. For example, see nested-if.cc:

```
#include <iostream>

using namespace std;

int main()
{
  int donation;
  cout << "Enter your donation amount in dollars: ";
  cin >> donation;

  if (donation > 300) {
    cout << "You are now a member of our Patron's club!" << endl;
    if (donation > 1000) {
      cout << "Our president will contact you personally to thank you." << endl;
    }
  }
  else {
    cout << "Thank you very much for your donation." << endl;
  }

}
```

# The else if statement

Sometimes we have a chain of conditions that need to be evaluated. The “else if” statement helps with these. The first condition satisfied will determine what grade is printed, and once a grade is printed, the other checks will not be tested.

```
grade.cc:

int main()
{
  int grade;
  cout << "Enter the student's numerical grade: ";
  cin >> grade;

  if (grade >= 90)
    cout << "A" << endl;
  else if (grade >= 80)
    cout << "B" << endl;
  else if (grade >= 70)
    cout << "C" << endl;
  else if (grade >= 60)
    cout << "D" << endl;
  else
    cout << "F" << endl;

}
```

## switch/case

Useful if there are only a few possibilities and we want to do something different for each one. Often used with the enum data type. Each section usually ends with a “break” statement, otherwise the code keeps executing as one goes down the cases in the switch statement. The “default” case catches any un-listed cases.

```
switch-case.cc:
int main()
{
  enum category { Faculty, Staff, Grad, Undergrad };
  int employee = Staff;

  switch(employee) {
    case Faculty:
      cout << "Your parking fee is $400." << endl;
      break;
    case Staff:
      cout << "Your parking fee is $350." << endl;
      break;
    case Grad:
      cout << "Your parking fee is $200." << endl;
      break;
    case Undergrad:
      cout << "Your parking fee is $100." << endl;
      break;
    default:
      cout << "Wrong input, employee category not recognized." << endl;
      break;
  }
}
```

## while loops

The while statement keeps executing a code block until a given condition is no longer fulfilled. Here's a countdown timer example that prints numbers from 10 to 0 in decreasing order:

while.cc:

```
int main()
{
  int counter = 10;

  while (counter >= 0) {
    cout << counter << endl;
    counter--;
  }

}
```

Program output:

```
10
9
8
7
6
5
4
3
2
1
0
```

## do...while loops

The do...while statement is like the while statement but it ensures the code block executes at least once.  In this example, the counter will print once even if the user enters a negative number.

```
do-while.cc:

int main()
{
  int counter;
  cout << "Where should I start the countdown? ";
  cin >> counter;

  do {
    cout << counter << endl;
    counter--;
  } while (counter >= 0);

}
```

# for loops

The for loop is the most commonly encountered loop in C++. We set a variable (usually an integer) to some initial value, we loop as long as some condition is met, and at each iteration, we typically increment or decrement the variable. The general syntax is:

```
for (var = initial_value;
exit condition executed at beginning of each loop;
statement executed at end of each loop) {
    code block executed each loop;
}
```

Example:

```
 for (int i=0; i<5; i++) {              // prints i from 0 to 4,
not 5
    cout << “i = “ << i << endl;
 }
```

# Nested for loops

You can have a for loop inside another for loop.  For example, we might want to print out all the elements of a matrix.

```
print-matrix.cc:

int main()
{
  int matrix[2][2] = { {0, 1}, {2, 3} };

  cout << "Printing 2x2 matrix:" << endl;

  for (int row=0; row < 2; row++) {
    for (int col=0; col < 2; col++) {
      cout << matrix[row][col] << " ";
    }
    cout << endl; // line break at end of row
  }

}
```

Output:

```
Printing 2x2 matrix:
0 1
2 3
```

## The continue statement

Inside a loop, “continue” hops back up to the beginning of the loop

```
continue.cc:

int main()
{
  int counter;
  cout << "Printing all odd numbers between 1 and 5, inclusive: " << endl;
  for (int i=1; i<=5; i++) {
    if (i%2 == 0) continue; // skip the even ones
    else cout << i << " ";
  }
  cout << endl;

}
```

## The break statement

We briefly encountered “break” above in the switch/case example.  It breaks out of a switch/case statement or a loop.  The example below also demonstrates a for loop with an empty middle statement (no simple condition is checked at the beginning of the loop).

```
break.cc:

int main()
{
  int counter;
  cout << "Find first odd number > 3 that's divisible by 3: " << endl;
  for (int i=4; ;i++) {
    if ((i%2 != 0) && (i%3 == 0)) { // not even, and divisible by 3
      cout << "Found it!  It's " << i << endl;
      break;
    }
  }

}
```

# Chapter 6: Functions

- The purpose of functions
- Defining vs declaring functions
- Sending values to functions
- Getting values from functions
- Pass-by-value
- References
- Inline functions

## The purpose of functions

The primary purpose of functions is for *code re-use*. If you need to repeat a certain piece of code more than a few times, it is much better to implement it as a function --- this allows the code to be defined just once, and then called as many times as needed. Then, if changes are required, only the one function needs to be changed, not many repeated instances of the code. This makes the code much easier to maintain.

## Example: The grade list

In Chapter 5 we looked at program flow control and the “if / else if / else” structure using the example grade.cc, in which the user typed a numerical score and the program printed out the corresponding letter grade according to a built-in table (90+ = A, 80+ = B, etc.).

Suppose we want to automate this procedure and have the program read a list of numerical scores and then print them out along with the corresponding grades. We'd probably store the scores in a file that could be read; let's skip this part for now and just write the scores into the program. It would be silly to repeat the score-to-grade translation code for every score in the list; instead, we put it into a function that can be called as many times as desired.

## gradelist.cc, Part I

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
  char grade(int score); // declare the function grade()

  vector<int> scores; // no parentheses if we want to make an empty vector
  // We could imagine reading the scores from a file and pushing them
  // into the scores vector one at a time, until the end of file is
  // reached.  For simplicity, here we'll just directly push back 3 values.
  scores.push_back(67);
  scores.push_back(93);
  scores.push_back(82);

  for (int i=0; i<scores.size(); i++) {
    int thisScore = scores[i];
    char thisGrade = grade(thisScore); // here we call function grade()
    cout << "Score " << thisScore << " is grade " << thisGrade << endl;
  }
}
```

# Analysis of Part I

- vector<int> scores;
  The numerical scores are put in a vector that holds integers. Writing it this way instead of, say, scores(3), sets up an empty vector and we can add however many scores to it, one at a time, with the push_back() function
- int thisScore = scores[i]; char thisGrade = grade(thisScore);
  Within the loop over scores, we use variables thisScore and thisGrade.  Note that they are declared as type int and char, respectively, inside this loop. This means these variables will only exist within the loop (their scope is restricted to the loop). That's fine because we don't need/want them outside the loop.
- We set the grade according to
  char thisGrade = grade(thisScore);
  This is where we call the function grade(), we pass it the current score as an argument, and it returns to us a single character representing the corresponding grade ('A', 'B', etc.)

## Function declaration

- Near the top of the program we have

```
int main()
{
    char grade(int score);
…
}
```

- This bit of code is known as the *declaration* of function grade(). It is also called the *function signature* or the *function prototype*. It tells the program we are about to use a function called grade. It takes as input a single value, an integer, and it returns as output a character (here, the grade 'A', 'B', 'C', 'D', or 'F'). The variable(s) used as input to the function are called the function *arguments* or *parameters*. It only matters how many and of what type: the names used for function arguments (here, score) are irrelevant in the declaration and they don't have to match those given subsequently in the function *definition* (see below).

# Scope of function declarations

In this example, we declared function grade() *inside* function main(). That means we can only use function grade() inside function grade() inside function main(), just like any variable declared inside function main() can only be used inside function main().

If we wanted to use grade() in other functions in the program (there aren't any other functions in this simple example, but there would be in a typical program), then we would need to declare grade() outside the scope of main(). To make grade() usable by the entire file grade.cc, we move it outside of main() like this:

```
char grade(int score);

int main()
 {
  vector<int> scores;
    …
 }
```

# gradelist.cc, Part II

```
/ Here we define the function grade()
// The function converts a numerical score into a letter
grade
char grade(int score)
{
  if (score >= 90)
    return('A');
  else if (score >= 80)
    return('B');
  else if (score >= 70)
    return('C');
  else if (score >= 60)
    return('D');
  else
    return('F');
}
```

## Function definition

- The above listing, Part II of gradelist.cc, is the *function definition*. It defines what the function actually does. The definition needs to be consistent with the declaration in the sense that they need to use the same name for the function, and they need to agree on how many arguments, what type they are, and what order they're in.
- The function definition can use different names for the function arguments than the function declaration (although this can be poor practice because it can get confusing). Nevertheless, the names in the first line of a function definition must be consistent with the body of the definition. Since we called the input integer 'score' in the first line of the definition, we stick to this name in the body of the function (e.g., if (score >= 90) ...).

## Analysis of Part II

- Note that our grade() function has all the bases covered: the final “else return('F');” ensures that no matter what score is input into the function, we always get a letter grade character out of it. If a function is supposed to return a character, then the programmer must ensure that the function will always return a character, no matter what input it has been passed.
- What if the programmer tries to mess with the function and provide it invalid input, like calling it with a floating-point number instead of an integer grade? For example, grade(90.3)? Because we declared and defined grade() to take integers, this will result in a compiler error when we try to compile the program. If we have non-integer scores, we would need to re-write the program to work with floats or doubles instead of integers.

## Functions with no arguments, and type void

- It's perfectly fine to have a function that takes no arguments; that just means the function does the same thing every time it's called. In such a case, one can either just have an empty argument list (e.g., func()) or replace the argument list with the keyword “void” (e.g., func(void)) in the declaration and definition. When calling the function, it would be called like “func()” with no arguments.
- Likewise, if the function doesn't return anything, we say it returns “void”.
- As an example, let's consider a function that prints some kind of banner (maybe just a simple row of asterisks). It doesn't need arguments for input, and it doesn't need to return anything. It just prints. Such a function is presented in example banner.cc.

## Example banner.cc

```
#include <iostream>

using namespace std;

int main()
{
  void banner(void);

  banner();
  cout << "Hello world!" << endl;
  banner();
}

void banner(void)
{
  cout << "************************************************************" << endl;
}
```

Program output:

```
************************************************************
Hello world!
************************************************************
```

## Functions with multiple arguments

- It's easy to generalize to the case of multiple arguments in a function: in the declaration and definition, just list the arguments one at a time, with their types.  Strictly speaking, one does not need to give the arguments names in the declaration, but it is common practice to do so.
- For example, consider a function that computes the hypoteneuse (the long side) in a right triangle, according to the Pythagorean theorem, c = sqrt($a^2$+$b^2$), where sqrt() is the square root function (defined in C++ by the cmath header file).

## Example hypoteneuse.cc

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
  double hypoteneuse(double a, double b);
  double a, b, c; // three sides of a right triangle
  cout << "Enter the length of side a of right triangle: ";
  cin >> a;
  cout << "Enter the length of side b of a right triangle: ";
  cin >> b;
  c = hypoteneuse(a, b);
  cout << "The length of the hypoteneuse c is " << c << endl;
  return 0;
}

double hypoteneuse(double a, double b)
{
  double c = sqrt(a*a + b*b);
  return c;
}
```

## Functions with default parameters

- Frequently, we will call a function with one (or more) of the arguments being the same from call to call.  In such a case, it can be more convenient to assign a default value to this argument.
- For example, we could use the ideal gas law, PV=nRT, to calculate the product of pressure times volume (PV) if we know the number of moles of gas (n), the gas constant (R), and the temperature in Kelvin (T).  We might typically assume 1 mole of gas (n=1), and just vary the temperature (T).  But we'd like to retain the flexibility to do computations with different numbers of moles of gas (n) on occasion.  Having a function expect a default value n=1 is the solution.

# Example gas-law.cc

```
#include <iostream>

using namespace std;

int main()
{
  double PV(double T, double n = 1.0); // default values here
  double PV1 = PV(298.0);
  cout << "PV for n=1 and T=298 is " << PV1 << " atm*L" << endl;
  double PV2 = PV(298.0, 2.0); // this time don't use default n
  cout << "PV for n=2 and T=298 is " << PV2 << " atm*L" << endl;
  return 0;
}

// function computes pressure-volume product PV = nRT, given T and n
// T is in Kelvin, n is in moles, and P*V is in atmospheres * Liters
double PV(double T, double n) // default values not here again
{
   double R = 0.08206; // in L*atm/(mol*K)
   double PV = n * R * T;
   return(PV);
}
```

Program output:

```
PV for n=1 and T=298 is 24.4539 atm*L
PV for n=2 and T=298 is 48.9078 atm*L
```

## Overloaded functions

- In the previous example, we saw how the same function could be called with different numbers of parameters, depending on whether some of the parameters are left at their default values or not
- *Function overloading* generalizes this concept: if we have two (or more) functions that behave similarly but act on different data types, we can call these functions by the same name, and the correct function will be called depending on the arguments that are passed.

Example format-print.cc:
```
#include <iostream>
#include <stdio.h> // for printf() below

using namespace std;

int main()
{
  void format_print(int a);
  void format_print(double a);
  int a = 42;
  double b = 3.1415926;
  format_print(a);
  format_print(b);
  return 0;
}

void format_print(int a)
{
  printf("%4d\n", a);  // print an integer within 4 spaces
}

void format_print(double a)
{
  printf("%4.2lf\n", a); // print a double within 4 spaces,
                         // and 2 digits after the decimal
}
```

Program output:
```
  42
3.14
```

## Recursion

Recursion is when a function calls itself. This can be very useful, but one also has to be very careful to make sure that the recursion eventually stops --- otherwise, the recursion could go on forever and the program will never stop running.

In the next example, we use recursion to compute the factorial of an integer. For example, 5! = 5*4*3*2*1 = 120. Since this is a simple sequence, we can evaluate the factorial of a number n by multiplying n by the factorial of (n-1). We just have to be sure to stop when we reach 1.

Note: this routine only works for modest-sized integers, otherwise we will overflow the integer we're storing the result in.

## Example of recursion (factorial.cc)

```
#include <iostream>

using namespace std;

int main()
{
  int factorial(int a);
  int x = 5;
  cout << x << " factorial is " << factorial(5) << endl;
  return 0;
}

int factorial(int a)
{
  if (a == 1) return 1;
  else return(a * factorial(a-1));
}
```

```
Output:
5 factorial is
120
```

## Function parameters are “pass by value”

- When a function is called, the *values* of variables are passed to the function, not the variables themselves. This means that any change to a variable inside the function does not change the variable in the calling function.

## Example pass-by-value.cc

```
include <iostream>

using namespace std;

int main()
{
  int square(int x);
  int x = 2;
  cout << "Before calling square(), x is " << x << endl;
  int x2 = square(x);
  cout << "The square is " << x2 << endl;
  cout << "After calling square(), x is " << x << endl;
  return 0;
}

int square(int x)
{
  x = x * x;
  return(x);
}
```

Program output:

```
Before calling square(), x is
2
The square is 4
After calling square(), x is 2
```

## References

Sometimes the “pass by value” convention in C++ is inconvenient. We occasionally *want* a change to a variable's value to be reflected in the calling function. In such cases, we can use a “reference”. You can think of a reference as a way of telling the compiler we don't want to pass a copy of the variable, but we want to pass the variable itself (this is sometimes called “pass by reference”).

This is trivially easy to accomplish: we just add an ampersand (&) after the type of the variable(s) we want to be a reference. We do this in the function delaration and also in the function definition.

## Reference example (swap.cc)

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

int main()
{
  void swap(int& x, int& y); // void swap(int &x, int &y); also works
  int a = 1, b = 2;
  cout << "a = " << a << " b = " << b << endl;
  swap(a,b);
  cout << "a = " << a << " b = " << b << endl;
  return 0;
}

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

Output:

```
a = 1 b = 2
a = 2 b = 1
```

## Inline functions

When a function is called, there is some computational overhead associated with it. For typical functions that are only called a modest number of times, this is not noticeable. However, if a small function is called many, many times, then the program will be slowed down by all the function calls.

Inline functions were created to handle this situation. The “inline” directive asks the compiler to expand the function right where it is called during compilation (“at compile time”), instead of issuing a function call while the program is running (“at runtime”). It is as if the programmer cut and pasted the function contents everywhere the function is called --- but the compiler does the dirty work for us.

A disadvantage of inline functions is that they cause the compiled code to become significantly larger; hence, they are only recommended for very short functions.

# Specifying inline functions

The inline function should be defined *before* where it is called. The function does not need to be declared.

Any functions defined inside a class definition are assumed to be inlined.

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

inline double half(double a)
{
  return(a/2.0);
}

int main()
{
  double x = 22.4;
  double y = half(x);
  cout << "x = " << x << " y = " << y << endl;
  return 0;
}
```

# Chapter 7: Pointers and Dynamic Memory Allocation

- Pointers
- Dynamic memory allocation: new/delete, malloc()/free()
- const and pointers, const and references
- Trapping failures to new/malloc()
- Common pointer problems
- Using pointers or references for greater efficiency in function calls
- NULL pointers

# Defining pointers

- A pointer is a variable that holds a memory address; typically this is the location in memory associated with some other variable
- For example, variable “x” may hold an integer, and that integer may be stored in main memory at some location such as (in hexidecimal) 0x329a. We could store that memory location (0x329a) in a special variable called a pointer (with some other name, like “x_ptr”). That way we'd know where variable “x” is located in memory
- Knowing the memory location of “x” gives us more control over “x”

## Pointer example

Example pass-pointer.cc:

```
#include <iostream>

using namespace std;

int main()
{
  int square(int* x);
  int x = 2;
  cout << "Before calling square(), x is " << x << endl;
  int x2 = square(&x);
  cout << "The square is " << x2 << endl;
  cout << "After calling square(), x is " << x << endl;
  return 0;
}

int square(int* x)
{
  *x = (*x) * (*x);
  return(*x);
}
```

Output:

```
Before calling square(), x is
2
The square is 4
After calling square(), x is 4
```

## Overview of the example

- The pass-pointer.cc example is the same as the pass-by-value.cc example in the previous chapter, except that we converted the function square() to take a *pointer* to an integer instead of an integer. That is, when we call square(), we are passing the memory location of the integer we want to square, not the integer itself.
- Because we have the memory location of the variable, we can manipulate it directly and change it in the subroutine --- just like a reference
- Original C did not have references; pointers were the original way to handle references (although they are more general and more powerful in some circumstances than references)
- Note that function calls involving pointers are *still* pass-by-value; the value we pass is now the value of the memory location

## Details of the example

- The function declaration is “int square(int* x)”.  The star indicates that we are passing not an integer, but a pointer to an integer.  Like the & sign denoting a reference, the * sign can go anywhere between the type (int) and the variable (x).  For example, “int square(int *x)” is also valid.
- The first line of the function definition also uses a star in the same way: “int square(int* x)”.
- When calling the function, we pass not the value of x, but the memory location of x.  In this context, an ampersand (&) takes the address of x.  Thus we call the function like this: “int x2 = square(&x);”
- Inside the function, because we passed a memory address, “x” now refers to the memory location holding the original variable “x”, not the original variable “x” itself.  In this example, we don't want to manipulate the memory address of x, we want to manipulate the value it holds.  We use “*x” to mean “the variable being held at memory location x”.
- Unlike the pass-by-value.cc example, this example does change the original variable x inside the function square(), because we are manipulating the original location of the data, not a copy of it

## More about memory addresses

Memory addresses are usually very long numbers expressed in hexadecimal. For example, code such as the following:

```
int main()
{
  int x = 2;
  int* x_ptr = &x;
  cout << "The memory location of x is: " << x_ptr << endl;
  return 0;
}
```

produces the output

```
The memory location of x is: 0x7fff5ad428ac
```

Memory locations are determined by the computer at runtime and may differ each time the program is run

## Sizes of pointers

As we just saw, pointers contain long memory addresses. They have to be large enough to hold these long memory addresses. Hence, the size of a pointer is frequently larger than the size of the data it points to. For example, although a character is only one byte (8 bits), a pointer to a character (or any other data type) will typically be 64-bits for a 64-bit operating system

## Uses for pointers

There are two main uses for pointers:

- Pointers are an alternative to references.  Because the syntax for references is simpler (no need for lots of asterisks everywhere), references are now preferred for this use
- Pointers allow one to dynamically allocate memory for arrays, and to access and traverse arrays

## Accessing arrays with pointers

Example print-array.cc:

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

int main()
{
  void print_array(int* x, int length);
  int x[3] = {2, 4, 8};
  print_array(x, 3);
  return 0;
}

void print_array(int* x, int length)
{
  cout << "Printing array:" << endl;
  for (int i=0; i<length; i++) {
    cout << x[i] << " ";
  }
  cout << endl;
}
```

Output:

```
Printing
array:
2 4 8
```

The name of an array *is* a pointer; we don't need to pass &x to print_array() because x is *already* a pointer

## Traversing arrays with pointers

We can use an alternative syntax in the print_array() function from the previous example:

```
void print_array(int* x, int length)
{
  cout << "Printing array:" << endl;
  for (int i=0; i<length; i++) {
    cout << *x++ << " ";
  }
  cout << endl;
}
```

Now instead of accessing element i of x via x[i], we use *x++. *x gets the value of the variable pointed to by x, and the ++ increments the pointer to point to the next item in the array (after *x is determined).  Variable i is only used to count.

## Dynamic memory allocation

In the last example, we made array x to contain 3 elements in main():

```
int x[3] = {2, 4, 8};
```

This is an example of *static memory allocation*; the compiler knows at compile time that an array of 3 integers is needed.

But what if we don't know how many integers we need until after the program has started running? Syntax like the following does *not* work:

```
int n=3;
int x[n] = {2, 4, 8};  // won't work
```

# Dynamic memory allocation

Dynamic memory allocation is a way to allocate memory (for, e.g., an array) whose size isn't necessarily known at runtime. We previously saw examples of using the Standard Template Library (STL) “vector” to do such a thing. Dynamic allocation of basic arrays may be more efficient if we have larger-sized arrays. To allocate an integer array of length n (where n is a variable), we do:

```
int n = 3;
 int* array = new int[n];
```

We can also use this style even if we do know the length at compile time, e.g.,

```
int* array = new int[3]; // also works
```

## Freeing memory

If we allocate an array like this:

```
int* ptr = new int[10];
```

then when we're done with it, we should free the memory using

```
delete[] ptr; // frees the memory pointed to by ptr
```

We must handle this step ourselves; the pointer itself will be deleted when it goes out of scope (e.g., at the end of a function it's defined in), but the memory it points to will not automatically be deleted --- unless we use “smart pointers” (see below).

## Allocating/deleting a single variable

We've seen how to dynamically allocate an array, and how to delete it. For a single variable, the corresponding syntax for a variable of type “Type” is:

```
Type* ptr = new Type;
delete ptr;
```

Note there are no brackets in the delete call.

# malloc() and free()

The “new/delete” syntax is the preferred C++ way to allocate and delete variables dynamically

The old way to do this in C was to use malloc() (memory allocation) and free(). Many programs still use malloc()/free(), so it's good to be familiar with them

malloc() takes a number of bytes to allocate. If we want to allocate, say, 3 integers, then we compute the number of bytes as (3*sizeof(int)). malloc() returns a pointer of type (char*). Usually we convert this pointer to the type we want with a “cast”, like this:

```
int* array = (int*) malloc(3*sizeof(int));
```

When we're done with the array, we free up the memory for other uses by passing the pointer to the function free(), like this:

```
free(array);
```

# Example of new/delete, free/malloc

Example dynamic-allocation.cc:

```
#include <iostream>
#include <malloc.h> // for malloc() and free() calls
below
using namespace std;

int main()
{
  void print_array(int* x, int length);
  int* arr1 = new int[3];
  int* arr2 = (int*) malloc(3*sizeof(int));
  for (int i=0; i<3; i++) {
    arr1[i] = i;
    arr2[i] = i*2;
  }
  print_array(arr1, 3);
  print_array(arr2, 3);
  delete[] arr1; // made with new, must use delete[]
  free(arr2);    // made with malloc, must use free()
  return 0;
}
```

```
void print_array(int* x, int length)
{
  cout << "Printing array:" << endl;
  for (int i=0; i<length; i++) {
    cout << *x++ << " ";
  }
  cout << endl;
}
```

Output:

```
Printing
array:
0 1 2
Printing
array:
0 2 4
```

## Pointers and const

We've seen before that “const” designates a variable as a constant: it does not change. When we talk about pointers and const, there are three possibilities:

const int* ptr – ptr is a pointer to a constant int. The pointer could change (it could point to something else), but the thing it points to can't change.

int* const ptr – ptr is a constant pointer to an int. The pointer cannot point to anything else. But the int it points to could change.

const int* const ptr – ptr is a constant pointer to a constant int. Neither ptr nor what it points to can change.

This is easiest to remember as follows: “const” modifies the type or variable name immediately to its right.

# const example

Our print_array() function does not change the values pointed to: hence, we could tell the compiler we are passing a pointer to constant ints.

Could we go on to make x a “const int* const x”? No, because we increment x inside print_array(). However, because C++ is call by value, x is only incremented *inside* print_array(), not outside it. So x in main() isn't changed anyway.

Example print-array3.cc:

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

int main()
{
  void print_array(const int* x, int
length);
  int x[3] = {2, 4, 8};
  print_array(x, 3);
  return 0;
}

void print_array(const int* x, int length)
{
  cout << "Printing array:" << endl;
  for (int i=0; i<length; i++) {
    cout << *x++ << " ";
  }
  cout << endl;
}
```

# Another const example

We *can* rework the previous example to make use of a constant pointer to a constant int, we just have to avoid the “*x++” call inside print_array().  That's easily accomplished (see right).

example print-array4.cc:

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

int main()
{
  void print_array(const int* const x, int length);
  int x[3] = {2, 4, 8};
  print_array(x, 3);
  return 0;
}

void print_array(const int* const x, int length)
{
  cout << "Printing array:" << endl;
  for (int i=0; i<length; i++) {
    cout << x[i] << " ";
  }
  cout << endl;
}
```

## When are const's worth it?

Example print-array4.cc looks basically the same as print-array.cc but it has const's everywhere.  Why bother?

Using const when appropriate is considered good programming practice because it gives an extra clue to the compiler about what should be allowed to change when.  In return for this extra information, the compiler will provide an error if the const rules are ever broken. This can be a way to track down bugs easier.

On the other hand, one must admit that keeping track of all the const's can be a bit of trouble for the programmer. The cost/benefit analysis of using them must ultimately be decided by the programmer.

## Passing arguments to functions as pointers or references for efficiency

We have seen that both references and pointers allow us a way to have the program modify the parameters passed to a function.

Often it is advantageous to use pointers or references as function arguments even if we do *not* need to change the values of the parameters in a function.

Because function calls in C++ are pass-by-value, all the arguments are copied when the function is called. If they are simple integers, doubles, etc., or reference variables or pointers, this is no big deal, it's just a few bytes per argument. But if one of the arguments is a data structure or class, there can be enormous overhead copying it before it is passed to the function. Hence, for such large data types as classes/structures, it is better to pass a reference or pointer, even if we don't intend to modify that argument.

# const and references

If we did want to pass an argument to a function as a reference for reasons of efficiency, but we didn't want the function to modify that argument, we could tell the compiler this by adding “const” to the argument, like so:

```
 void compute_something(const int& number);
```

Note that the original variable “number” in the calling function need not be a “const int” --- we are only treating it as a const int within this function. C++ also allows us to declare the function like this:

```
 void compute_something(int& const number);
```

which will be equivalent for references.

# Pointer problems

Pointers are very powerful, but that power can easily get a programmer into trouble. Debugging problems with pointers is the hardest thing about programming in C++. Here are some tips to keep in mind:

Always free dynamically allocated memory when you're done with it. Failure to do this means the program starts eating up more and more memory: a *memory leak*. Use “delete” after “new,” and free() after malloc() --- and never mix these up.

Never free memory more than once. If you have two pointers pointing to the same chunk of allocated memory, only call delete/free on one of them.

Never attempt to free memory you never actually wound up allocating (for example, this can happen if the allocation is inside an “if” statement but the code to free the memory is outside the “if”).

Never attempt to access memory beyond that actually allocated. If you created an array of length 3, don't try to access the fourth element.

Don't try to access a location in memory after it has been freed with delete/free.

## Memory corruption

If pointers are not handled properly, it is possible to wind up with a pointer pointing to memory it shouldn't point to. If the program writes to this memory, in the best case, the program catches that something is wrong, and we get a segmentation fault or other error. In the worst case, the write actually occurs --- but with unpredictable results! Suddenly, an array that held perfectly good data now becomes corrupted because new data has been written into it by accident, when the new data was supposed to go into some other memory location.

Such a situation gives rise to “non-local” effects, when one part of the program that used to work perfectly now no longer does, because the data it uses got corrupted by a pointer going wrong in some completely different part of the code. This can be very hard to debug. Debugging tools like “valgrind” are useful in such situations.

# NULL pointers

To help avoid some of the common mistakes with pointers, it's good to initialize all pointers to NULL if we are not immediately assigning them a valid memory address when they are defined. This is a good idea because a definition like this:

```
int* ptr;  // bad --- could point anywhere until address assigned
           // (and we might forget to assign it a valid address!)
```

creates a pointer that points to some random memory location. If we use it before we assign it a value, all sorts of mayhem can ensue. It is safer in a situation like this to assign the pointer immediately to NULL; then we can easily check whether the pointer points to a valid memory address, or whether it points to NULL.

```
int * ptr = NULL; // good --- we can check for NULL before
                  // we try to use the pointer
```

# What if dynamic allocation fails?

If the computer is low on available memory, then dynamic memory allocation (whether via new or free) might fail. If this happens, our program will crash soon thereafter, unless we can detect the failure to allocate, and handle it gracefully.

In C-style malloc(), if malloc() fails, it returns NULL. So, we can check like this:

```
 double *array;

 if ((array = (double *) malloc(length*sizeof(double))) == NULL)
{
   fprintf(stderr,"init_array: trouble allocating memory \n");
   fprintf(stderr,"size = %ld\n",size);
   // do something to abort here
 }
 else return(array);
```

## Handling failure of “new”

If we use the more modern “new” to create memory dynamically, we can catch allocation failures using “exception handling” and the try/catch structure.  std::bad_alloc is a pre-defined exception.  More on exception handling later.

Example catch-new-failure.c:

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

int main()
{
  try {
    int* ptr = new int[9982381213]; // too big?
    cout << "Created memory." << endl;
    delete[] ptr;
  }
  catch (bad_alloc) {
    cout << "Failure to allocate memory." << endl;
  }
  return 0;
}
```

## new(nothrow)

If this exception handling business seems to complicated for now, one can alternatively tell new to return NULL (like malloc()) instead of throwing an exception.

Example new-nothrow.cc:

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

int main()
{
  int* ptr = new(nothrow) int[9982381213]; // too big?
  if (ptr != NULL) {
    cout << "Created memory." << endl;
    delete[] ptr;
  }
  else
    cout << "Failure to allocate memory." << endl;

  return 0;
}
```

## Smart pointers

Keeping track of whether or not we need to free dynamically allocated memory, and when, can be a bit of a pain. That's why “smart pointers” were created.

A smart pointer is a C++ class that tracks when it is ok to free dynamically allocated memory, and it facilitates the automatic freeing of such memory at the appropriate time.  Otherwise, it acts like a regular pointer.

This incredibly useful technique is most often associated with classes, so we will defer further discussion to Part II of the notes.

# Chapter 8: Streams

- Stream basics
- Formatting std::cout output
- Reading variables and strings from std::cin
- File streams
- Writing a file
- Reading a file
- Stringstream

## Stream basics

Our first example was the “Hello, world!” program, which prints out a simple message:

```
std::cout << “Hello, world!” << std::endl;
```

This is already an example of streams... std::cout is an output stream that prints to “standard output,” i.e., the screen. Similarly, std::cin is an input stream that reads from the keyboard:

```
std::cin >> number;
```

Streams in C++ are a generic way to handle input and output. The stream insertion operator, <<, can be used to write to the screen, to a file, to a device, etc.  Likewise, the stream extraction operator, >>, can be used to read from the keyboard, from a file, etc.  We just replace std::cin and std::cout with whatever stream we need!

## Popular C++ Streams

std::cout – standard output, usually prints on the screen / terminal window

std::cin – standard input, usually reads from the keyboard

std::cerr – standard output stream for errors, usually prints on the screen like cout, but could be “redirected” elsewhere

std::fstream – input/output for files

std::ofstream – output stream for files

std::ifstream – input stream for files

std::stringstream – read/write from/to strings; useful for formatting strings or using strings for data conversions

## Formatting cout output

We can format the output coming out of cout in various ways. We do this by using “manipulators” on the stream. We already know about std::endl, which inserts a newline character. Other manipulators are:

- dec – interpret as decimal
- hex – interpret as hexadecimal
- oct – interpret as octal
- fixed – use fixed-point notation (default)
- scientific – use scientific notation (e.g., 6.626E34)

## More manipulators

The following additional manipulators are available if we #include <iomanip>:

- setprecision – give number of digits to print after decimal
- setw – set the width of the field to print in
- setfill – set a character to fill in empty space
- setbase – set the base, like using dec/hex/oct in previous slide
- setiosflag – set flags using a bitwise mask; available flags are of type std::ios_base::fmtflags.
- resetiosflag – restore default values to flags set by setiosflag

## Formatted cout example 1

Example print-numbers.cc:

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

int main()
{
  int x = 255;
  cout << "decimal: x = " << x << endl;
  cout << "octal  : x = " << oct << x << endl;
  cout << "hex    : x = " << hex << x << endl;
  cout << setiosflags(ios_base::hex|ios_base::showbase|ios_base::uppercase);
  cout << "In hex with base notation: x = " << x << endl;
  cout << resetiosflags(ios_base::hex|ios_base::showbase|ios_base::uppercase);
  cout << "After resetting flags: x = " << x << endl;
}
```

Program output:

```
decimal: x = 255
octal  : x = 377
hex    : x = ff
In hex with base notation: x = 0XFF
After resetting flags: x = 255
```

# Formatted cout example 2

Example print-numbers2.cc:

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

int main()
{
  double h = 6.62606957E-34; // Planck's constant in J*s
  cout << "h = " << h << endl;
  cout << "Setting precision to 4 digits after the decimal" << endl;
  cout << setprecision(4);
  cout << "h = " << h << endl;
  cout << "Print h (right-aligned) in a field 12 spaces long" << endl;
  cout << "h = " << setw(12) << h << endl;
  cout << "Note: setprecision() affects future numbers sent to cout" << endl;
  cout << " but setw() only affects the very next thing sent to cout" << endl;
}
```

Program output:

```
h = 6.62607e-34
Setting precision to 4 digits after the decimal
h = 6.626e-34
Print h (right-aligned) in a field 12 spaces long
h =    6.626e-34
Note: setprecision() affects future numbers sent to cout
 but setw() only affects the very next thing sent to cout
```

## cin example

We can use std::cin to read plain old data types from the keyboard. It also reads strings … however, it assumes whitespace characters (e.g., a space) begin a new string. To read a string with spaces in it, we need to use getline(cin, string).

Example cin.cc:

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

int main()
{
  int i;
  string a;
  cout << "Enter an integer:" << endl;
  cin >> i;
  cout << "You entered " << i << endl;
  cout << "Enter a string with no spaces:" << endl;
  cin >> a;
  cout << "You entered " << a << endl;
  cout << "Enter a string with spaces:" << endl;
  getline(cin, a);
  cout << "You entered " << a << endl;
}
```

Example program input/output:

```
Enter an integer:
32
You entered 32
Enter a string with no spaces:
David Sherrill
You entered David
Enter a string with spaces:
You entered  Sherrill
```

## File streams

We can read and write files with streams in a very similar fashion as we use cin to read from the keyboard or cout to write to the screen.

1. #include <fstream>

2. create a file stream variable, like this:
```
fstream outFile;
```

3. Open the file by providing the filename and any arguments saying whether this is for input, output, or both; if the file is human-readable text (default) or non-human-readable binary format (allowing more compact files), whether the new file should delete any existing file of this filename (“trunc” option) or whether it should append onto existing files (“app” option), etc.  For example:
```
outFile.open(“output.dat”, ios_base::out|ios_base::trunc)
```

4. Alternatively, steps 2+3 can be combined in a constructor like this:
```
fstream outFile(“output.dat”, ios_base::out|ios_base::trunc);
```

5. Make sure the file is open before reading/writing:
```
if (outFile.is_open()) {
   // do stuff
   outFile.close(); }
```

## File stream options

- ios_base::in – open file for reading
- ios_base::out – open file for writing
- ios_base::binary – open binary file (text is the default)
- ios_base::trunc – delete any file that might already exist with this name (default)
- ios_base::app – append to the end of any existing file with this name
- ios_base::ate – start working at the bottom of the file

Note: by default, files will be open for both read *and* write access, if neither ios_base::in nor ios_base::out are specified.

# Writing to a text file

Example writefile.cc:

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

int main()
{
  ofstream outFile;
  outFile.open("summary.txt", ios_base::out);
  if (outFile.is_open()) {
    cout << "Beginning to write file summary.txt" << endl;
    outFile << "Here is your summary:" << endl;
    outFile << "Everything is working well today!" << endl;
    outFile.close();
    cout << "Done writing to file!" << endl;
  }

  return 0;
}
```

This creates a text file called “summary.txt” in the current working directory and prints a couple of lines to it.

# Reading a text file

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

int main()
{
  ifstream inFile;
  // read the file we wrote in writefile.cc
  inFile.open("summary.txt", ios_base::in);
  if (inFile.is_open()) {
    cout << "Beginning to read file summary.txt" << endl << endl;
    string lineIn;
    while (inFile) {
      getline(inFile, lineIn); // read entire line at a time
      if (lineIn.length() > 0) // final read attempt will return a blank line
        cout << lineIn << endl;  // print line if not empty
    }
    inFile.close();
    cout << endl << "Done reading from file!" << endl;
  }
  else {
    cout << "Error: Could not open file for reading." << endl;
  }
  return 0;
}
```

## Reading beyond end-of-file

- Notice in the previous example that or final getline returns a blank line (with length 0). After this happens, inFile returns false on the next test of while(inFile). We need to remember to do special handling of this blank line whenever we use getline inside of a “while(inFile)” block
- Perhaps surprisingly, if we were to keep trying to read additional lines with getline(), we would keep getting more blank lines, rather than an I/O error
- Similarly, if a file open fails and we try to use getline(), we will also just get blank lines

## C-style file I/O

The C++ tools are perfectly adequate for reading and writing files, but so are the older, C-style functions. You may encounter the C-style I/O functions if you work with an older code base. The syntax is different, but the C functions are fairly analogous to the C++ functions.

## Stringstream

Stringstream is a special stream that can read and write basic data types (integers, doubles, etc.) as well as strings.  This allows one to convert basic data types to/from strings (e.g., an integer to a string, or vice versa).

To use stringstream, #include <sstream>

To create a stringstream,
stringstream sStream;

The next example shows how to convert a double-precision number to a string, and vice-versa, using stringstream

Example stringstream.cc:

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

int main()
{
  stringstream stream1, stream2;
  double x = 1.29312, y;
  string a;

  // convert double into a string
  stream1 << x; // push double x into the stringstream
  stream1 >> a; // pull x out of the stringstream into a string
  cout << "String = " << a << endl; // print the string (NOT the stringstream)

  // convert string back to a double... need to use *new* stringstream
  stream2 << a; // push string onto stringstream
  stream2 >> y;
  cout << "Double = " << y << endl;

  return 0;
}
```

```
// stringstream2.cc
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
  int t = 4, result = 42;
  stringstream ss;

  ss << "Try " << t << " gave a value of " << result;
  cout << ss.str() << endl; // convert to string for printing

  // Let's reuse the stringstream.  Reset it by replacing
  // the string it contains with a blank one.
  ss.str(std::string());

  ss << "Hello, world!";
  cout << ss.str() << endl;
}
```

Program output:

```
Try 4 gave a value of 42
Hello, world!
```