C++ Primer

(Based upon Objects, Abstraction, Data Sturctures and Design Using C++)

Primer Chapter Outline

- The C++ Environment
- Preprocessor Directives and Macros
- C++ Control Statements
- Primitive Data Types and Class Data Types
- Objects, Pointers, and References
- Functions
- Arrays and C Strings
- The `string` Class
- Input/Output Using Streams

Moving from Java to C++...

"Java is C++ without the guns, knives, and clubs."
- James Gosling, Creator of Java

Originally there was C, developed from 1969-1973 in parallel with Unix

C is a small language used to write low-level software such as device drivers, OS kernels (e.g., Linux), and compilers for languages such as Java. It is a *subset* of C++

C++ was developed from 1983-1985 as an extension of C to include object-oriented programming (OOP).

Java is based heavily on C++

Moving from Java to C++...

	Java	C++
char	Unicode	8-bit number (possibly unsigned)
Arrays	A class with different properties (e.g. has a length member)	No length! Doesn't check for out of bounds
struct	No	Yes
>>>	Yes	No
::	No	Yes, after classes. Member functions can be implemented outside of classes
	Uses packages	Uses namespaces
if condition	boolean only	int, char, bool
classes	everything!	global declarations and functions
object construction	always via new	`class Point { ... };` `Point p(2.1, 1.5), q;`
garbage collection	automatic!	must create destructors and delete objects created by new

Moving from Java to C++...

- Note:
  - Within conditionals (if and loop guards), C++ allows integer expressions, where 0 evaluates to FALSE, and non-0 evaluates to TRUE)
    - E.g. `if( someInt != 0)`
  - This is poor style as it is unclear:
    - `if( someInt )`
  - Common error(!):

```
int a = 0, b = 1;
if( a = b) { ... }
```


Moving from Java to C++...

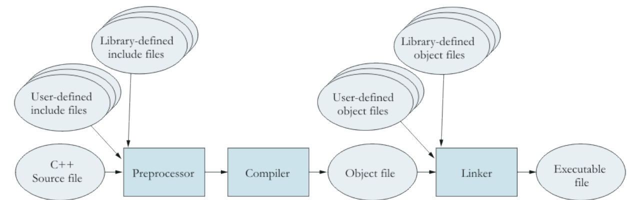
Basic Types

Java	C/C++
boolean	bool
char	char/wchar_t
byte	char/wchar_t
int	int
float	float
double	double

Compiling and Linking

- A C++ program consists of one or more source files.
- Source files contain function and class declarations and definitions.
  - Files that contain only declarations are incorporated into the source files that need them when they are compiled.
    - Thus they are called *include files*.
  - Files that contain definitions are translated by the compiler into an intermediate form called *object files*.
  - One or more object files are combined with to form the executable file by the linker.

Compiling and Linking



A Simple Program

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

using namespace std;

int main()
{
  cout << "Enter your name\n";
  string name;
  getline(cin, name);
  cout << "Hello " << name
      << " - welcome to C++\n";
  return 0;
}
```


Unix & C++

- Unix commands are entered on the command line, after the ">" symbol. Common Unix commands include:

Listing files: `ls -l`

Listing your current (working) directory: `pwd`

Copying and removing files:
```
cp filename secondFileName
rm filename2
```

Renaming (moving) files:
```
mv filename myNewFileName
```

Directory operations:
```
cd directoryName       (change to subdir)
mkdir directoryName    (create directory)
rmdir directoryName    (remove directory)
```

There are many introductory references on Unix, both online and in print. Please refer to those for more information.

Compile, Link & Run:

`g++ <options> <files>`

Useful options:

`-o <executable name>` (default `a.out`)

`-Wall` (turn on all warning messages)

Type "`man g++`" or "`man gcc`" to see the online *manual page* for GNU C++

E.g. `g++ -Wall myProgram.cpp` will produce `a.out` which can be run with `./a.out` (or just `a.out`, depending on your environment settings)

Alternatively: `g++ -Wall -o myProg myProgram.cpp` will produce `myProg` which can be run with `./myProg`

The using Statement

- The line
  `using namespace std;`
  tells the compiler to make all names in the predefined namespace std available.
- The C++ standard library is defined within this namespace.
- Incorporating the statement
  `using namespace std;`
  is an easy way to get access to the standard library.
  - But, it can lead to complications in larger programs.

The `using` declaration

- Instead of incorporating all names from a namespace into your program
  - It is a better approach to incorporate only the names you are going to use.
  - This is done with individual using declarations.

```
using std::cin;
using std::cout;
using std::string;
using std::getline;
```


The function `main`

- Each program must include a main function.
- This function is defined as follows:

```
int main()
{
  …
}
```

where the code for the function appears between the { and the }.

The stream insertion operator

- The statement:

```
cout << "Enter your name\n";
```

inserts the string into the standard output stream.

  - The result is that it is displayed on the console.

The `getline` function

- The statement

```
getline(cin, name);
```

reads the characters from the input stream (keyboard) until a new line character is entered.

  - The resulting string is stored in the `string` `name`.

The insertion operator again

- The statement:

```
cout << "Hello " << name << " – welcome to C++\n";
```

outputs three strings to the console:

Hello
the entered line
– welcome to C++

- If the characters John Doe were entered, the result would be

Hello John Doe – welcome to C++

Splicing Long Lines

- If a line ends with the character \ (or the trigraph sequence ??/)
  - Then the following line is appended to this line and the result is considered a single line.

Comments

- Same as Java (minus JavaDOC)

Function Definition

- Form:

  return-type function-name(*parameter list*) {
  function body
  }

- The parameter list is either empty, or a comma-separated list of the form:

  type-name parameter-name

- Function definitions are generally placed in their own file, or related function definitions may be grouped into a single file.

Function Declaration

- To use a function within a source file before its full definition it must be declared as a protoype.
- Form:

  return-type function-name(*parameter list*);

  Within the parameter list, only the types of the parameters are required.

```
char min_char(char, char);
```


Common Errors:

syntax error
(look for a missing semi-colon)

undeclared function
(look for a misspelled keyword or a missing prototype)

unterminated string
(look for a missing quote)

undeclared identifier
(declare the identifier's type)

parse error
(probably missing a brace (curly bracket): { or }

Arrays

- An array is an object.
- The elements of an array are all of the same type.
- The elements of an array are accessed by an index applied to the subscript operator.

  array-name[*index*]

Declaring an array

- Form:

  type-name array-name[*size*];
  type-name array-name[] = {*initialization list*};

- Examples:

```
int scores[5];
string names[] = {"Sally", "Jill", "Hal", "Rick"};
```

Using braces and indentation

- There are several coding styles.
- The one used in this text is:
  - Place a { on the same line as the condition for an **if**, **while**, or **for** statement.
  - Indent each line of the controlled compound statement.
  - Place the closing } on its own line, indented at the same level as the if, while, or for.
  - For else conditions, use the form:

```
} else {
```


String Constants

- The form "*sequence of characters*"

  where sequence of characters does not include '"' is called a string constant.
- Note escape sequences may appear in the sequence of characters.
- String constants are stored in the computer as arrays of characters followed by a '\0'.

Increment and Decrement

- Prefix:

  ++x

  x is replaced by x+1, and the value is x+1

  --x

  x is replaced by x-1, and the value is x-1
- Postfix

  x++

  x is replaced by x+1, but the value is x

  x--

  x is replaced by x-1, but the value is x

Prefix and Postfix Increment (2)

- Assume that i has the value 3.
- Then

  z = ++i;

  would result in both z and i having the value 4.
- But

  z = i++;

  would result in z having the value 3 and i the value 4.

Automatic Type Conversion

- If the operands are of different types, the following rules apply:
  - If either operand is long double, convert the other to long double.
  - If either operand is double, convert the other to double.
  - If either operand is float, convert the other to float.
  - Convert char and short to int
  - If either operand is long, convert the other to long.

Explicit Type Conversion

- An expression of one primitive type can be converted to another primitive type using the form:

  new-type(*expression*)
- Example
  - If i is an int and y a double

    i = int(y);

    will convert y to an int and store int into i.
  - The statement:

    i = y;

    will do the same thing, but may result in a warning message.

The Conditional Operator

- Form:

  boolean-expression ? *value1* : *value2*

  If the *boolean-exression* is true, then the result is *value1* otherwise it is *value2*.
- In most cases the same effect can be achieved using the **if** statement.

Objects, Pointers, References

- An *object* is an area of computer memory containing data of some kind.
- The kind of data is determined by the object's *type*.
- A type may be either
  - A primitive type.
  - A user-defined (class) type.
- For class types
  - Objects may be contained within other objects.

Object Declaration

- Form:

  type-name name;

  type-name name = *initial-value*;

  type-name name(*argument-list*);
- Example

```
int i;
string s = "Hello";
double x = 5.5;
double y(6.7);
point p(x, y);
```


Object Lifetimes

- Objects are created when they are declared.
- Objects declared within the scope of a function are destroyed when the function is exited.
- Objects declared in a block (between { and }) are destroyed when the block is exited.
- Objects declared outside the scope of a function (called global objects)
  - Are created before main is called
  - Are destroyed after main exits
- Objects created using the **new** operator must be destroyed using the **delete** operator.

Pointers

- A pointer is an object that refers to another object.
- A pointer object contains the memory address of the object it points to.
- Example:

```
double x = 5.1234;
double *px = &x;
```

Pointer Declaration

- Form

  type-name pointer-variable*;

  type-name pointer-variable* = &*object*;

  *type-name *pointer-variable*;

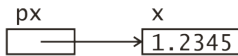
  *type-name *pointer-variable* = &*object*;

The dereferencing Operator

- The unary operator * is the *dereferencing* operator.
  - It converts a pointer to the value pointed to.
- Example:

  *px = 1.2345;

  Results in



Multiple Variables in one Declaration

- The declaration:

  ```
  double* px, py;
  ```

  declares that px is a pointer-to-double, but py is a double.
- To declare multiple pointer variables in one declaration:

  ```
  double *px, *py;
  ```


The NULL pointer

- The null pointer is a pointer value that points to nothing.
- Internally the value of the null pointer is implementation defined.
- The literal constant 0 is converted to a null pointer.
- Null pointers are converted to **false** when used in boolean expressions, and non-null pointers are converted to **true**.
- The macro NULL is defined in `<cstddef>` as:

  ```
  #define NULL 0
  ```
- Future versions of C++ will have a reserved-word for the null pointer literal.

The new operator

- Pointers are not generally initialized using the address-of operator.
- The new operator will create an instance of an object and return the address.

  ```
  double* px = new double;
  *px = 5.1234;
  ```


The delete operator

- All objects that are dynamically created using the new operator must be destroyed using the delete operator.

  delete *pointer-variable*;
- Note that pointer-variable must have been initialized by the new operator.
- Attempts to use delete on some other pointer value will probably cause a run-time error.

Call by reference vs. value

- By default, functions are called by value.
  - A copy of the arguments are made and stored into objects corresponding to the parameters.
  - Any changes made to the parameter values do not affect the original argument objects.
- If a parameter is declared to be a reference type, then:
  - The parameter variable is bound to the argument value.
  - Any change made to the parameter value is made to the original argument object.

Example of call by reference

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

- The statement:

```
swap(i, j);
```

will result in the values stored in i and j to be exchanged.

Call by const reference

- Class types may occupy several storage locations in memory.
- Passing a class type object by value is inefficient.
- By declaring the parameter to be a const reference, function can access the value of the argument, but not change it.

Example of const reference

```
int count_occurences(char c, const string& s) {
  int count = 0;
  for (int i = 0; i < s.size(); i++) {
    if (c == s[i]) count++;
  }
  return count;
}
```


Pointers and Arrays

- C++ performs automatic conversion between array types and pointer types.
- The expression:
  `students[0]`
  and
  `*students`
  are equivalent.
- The expression:
  `a[i]`
  is equivalent to
  `*(a + i)`
  and
  `&a[i]`
  to
  `(a + i)`

Dynamically Allocated Arrays

- The new[] operator can be used to allocate an array.
- Form:
  new *type-name*[*size*]

  will allocate space for size objects of type type-name are return a pointer to the first object.
- A declaration of the form:
  pointer-variable = new *type-name*[size];

  will initialize pointer-variable to point to the dynamically allocated array.
  - The pointer-variable can then be used like an array variable.

The delete[] operator

- All dynamically allocated arrays must be destroyed using the delete[] operator.
- Form

  delete[] *pointer-variable*;

  Note that *pointer-variable* must have been initialized using the new[] operator.

Arrays as function arguments

- Arrays are passed as pointers to functions.
- Function parameters may be declared either as pointers or arrays,
  - but the two are equivalent.
- Example:

```
int find(int x[], int n, int target);
int find(int* x, int n, int target);
```

are equivalent.

- You can call this funcion with either an array or a pointer:

```
int loc = find(scores, 10, 50);
int loc = find(scores + 5, 5, 50);
```


C-Strings

- The C programming language uses an array of char values terminated with the null character ('\0').
- Thus the constant

```
"hello"
```

is stored as:

h	e	l	l	o	\0

The string class

- The `string` class is defined in the header `<string>`
- Using the string class allows us to manipulate string objects similar to objects of the primitive types.
- Example:

```
string s1, s2;
s1 = "hello";
s2 = s1 + " world";
```


Input/Output Streams

- An input stream is a sequence of characters.
  - They may be from the keyboard, a file, or some other data source (e.g. a network socket).
- An output stream is a sequence of characters.
  - They may be written to the console, a file, or some other data source (e.g. a network socket).

The <iostream> header

- The header `<iostream>` declares the following pre-defined streams as global variables:

```
istream cin;  //input from standard input
ostream cout; //output to standard output
ostream cerr; //output to the standard error
```

- Standard input is generally from the keyboard, but may be assigned to be from a file.
- Standard output and standard error are generally to the console, but may be assigned to a file.

The istream class

- The istream class performs input from input streams.
- It defines the extraction operator (>>) for the primitive types and the string class.

Type of operand	Processing
`char`	The first non-space character is read.
`string`	Starting with the first non-space character, characters are read up to the next space.

`int`
`short`
`long`
	If the first non-space character is a digit (or + or -), characters are read until a non-digit is encountered. The sequence of digits is then converted to an integer value of the specified type.

`float`
`double`
`long double`
	If the first non-space character is a digit (or + or -), characters are read as long as they match the syntax of a floating-point literal. The sequence of characters is then converted to a floating-point value of the specified type.

Status Reporting Functions

Member Function	Behavior
`bool eof() const`	Returns **`true`** if there is no more data available from the input stream, and there was an attempt to read past the end.
`bool fail() const`	Returns **`true`** if the input data did not match the expected format, or if there is an unrecoverable error.
`bool bad() const`	Returns **`true`** if there is an unrecoverable error.
`bool operator!() const`	Returns `fail()`. This function allows the `istream` variable to be used directly as a logical variable.
`operator void*() const`	Returns a null pointer if `fail()` is `true`, otherwise returns a non-null pointer. This function allows the use of an `istream` variable as a logical variable.

Reading all input from a stream

```
int n = 0;
int sum = 0;
int i;
while (cin >> i) {
  n++;
  sum += i;
}
if (cin.eof()) {
  cout << "End of file reached\n";
  cout << "You entered " << n << numbers\n";
  cout << "The sum is " << sum << endl;
} else if (cin.bad()) {
  cout << "Unrecoverable i/o error\n";
} else {
  cout << "The last entry was not a valid number\n";
}
```


The ostream class

- The ostream class provides output to an output stream.
- It defines the insertion operator (<<) for primitive types and the string class.

Type of operand	Processing
`char`	The character is output.
`string`	The sequence of characters in the string is output.
`int` **`short`** **`long`**	The integer value is converted to decimal and the characters are output. Leading zeros are not output unless the value is zero, in which case a single 0 is output. If the value is negative, the output is preceded by a minus sign.
`float` **`double`** **`long double`**	The floating-point value is converted to a decimal representation and output. By default a maximum of six digits is output. If the absolute value is between 10^{-4} and 10^{6}, the output is in fixed format; otherwise it is in scientific format.

Formatting Manipulators in <iostream>

Manipulator	Default	Behavior
`noshowpoint`	yes	If a floating-point value is a whole number, the decimal point is not shown.
`showpoint`	no	The decimal point is always shown for floating-point output.
`skipws`	yes	Sets the format flag so that on input white space (space, newline, or tab) characters are skipped.
`noskipws`	no	Sets the format flag so that in input white space (space, newline, or tab) characters are read.
`right`	yes	On output, the value is right-justified.
`left`	no	On output, the value is left-justified.
`dec`	yes	The input/output is in base 10.
`hex`	no	The input/output is in base 16.
`fixed`	no	Floating-point output is in fixed format
`scientific`	no	Floating-point output is in scientific format.
`ws`	no	On input, whitespace is skipped. This is a one-time operation and does not clear the format flag.
`endl`	no	On output, a newline character is written and the output buffer is flushed.

I/O Manipulators in <iomanip>

Manipulator	Behavior
`setw(size_t)`	Sets the minimum width of the next output. After this the minimum width is reset to the default value of 0.
`setprecision(size_t)`	Sets the precision. Depending on the output format, the precision is either the total number of digits (scientific) or the number of fraction digits (fixed). The default is 6.
`setfill(char)`	Sets the fill character. The default is the space.
`resetiosflags(ios_base::fmtflags)`	Clears the format flags set in the parameter.
`setiosflags(ios_base::fmtflags)`	Sets the format flags set in the parameter.

Floating-point output format

- The default floating-point format is called general.
- If you set either fixed or scientific, then to get back to general format you must use the mainiplator call:

resetiosflage(ios_base::fixed | ios_base::scientific)

Format	Example	Description
Fixed	`123.456789`	Output is of the form `ddd.ffffff` where the number of digits following the decimal point is specified by the precision.
Scientific	`1.2345678e+002`	Output is of the form `d.ffffff±ennn` where the number of digits following the decimal point is controlled by the value of precision. (On some systems only two digits for the exponent are displayed.)

General	`1.23456e+006`
	`1234567`
	`123.4567`
	`1.234567e-005`
		A combination of fixed and scientific. If the absolute value is between 10^{-4} and 10^{6}, output is in fixed format; otherwise it is in scientific format. The number of significant digits is controlled by the value of precision.

File Streams

- The header <fstream> defines the classes

  ifstream An istream associated with a file

  ofstream An ostream associated with a file

Constructors and the open function

Function	Behavior
`ifstream()`	Constructs an `ifstream` that is not associated with a file.
`ifstream(const char* file_name, ios_base::openmode mode = ios_base::in)`	Constructs an `ifstream` that is associated with the named file. By default, the file is opened for input.
`ofstream()`	Constructs an `ofstream` that is not associated with a file.
`ofstream(const char* file_name, ios_base::openmode mode = ios_base::out)`	Constructs an `ofstream` that is associated with the named file. By default, the file is opened for output.
`void open(const char* file_name, ios_base::openmode)`	Associated an `ifstream` or and `ofstream` with the named file and sets the openmode to the specified value.

Openmode Flags

openmode	Meaning
`in`	The file is opened for input.
`out`	The file is opened for output.
`binary`	No translation is made between internal and external character representation.
`trunc`	The existing file is discarded and a new file is written. This is the default and applies only to output.
`app`	Data is appended to the existing file. Applies only to output.

String Streams

- Defined in the header <sstream>
- Associates an istream or ostream with a string object.

Constructor	Behavior
`explicit istringstream(const string&)`	Constructs an `istringstream` to extract from the given string.
`explicit ostringstream(string&)`	Constructs an `ostringstream` to insert into the given string.
`ostringstream()`	Constructs an `ostringstream` to insert into an internal string.
Member Function	**Result**
`string str() const`	Returns a copy of the string that is the source or destination of the `istringstream` or `ostringstream`.

Using an istringstream

- Assume that the string person_data contains:

  Doe, John 5/15/65

- We want to split this into family_name, given_name, month, day, and year.

```
istringstream in(person_data);
in >> family_name >> given_name;
in >> month; // Read the month
in >> c;     // Skip the / character
in >> day;   // Read the day
in >> c;     // Skip the / character
in >> year;  // Read the year
```


Using an ostringstream

- We want to construct the string person_data from the component values.

```
ostringstream out;
out << family_name << ", " << given_name << " "
  << month << "/" << day << "/" << year;
string person_data = out.str();
```

The #include Directive

- The first two lines:

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

incorporate the declarations of the iostream and string libraries into the source code.
- If your program is going to use a member of the standard library, the appropriate header file must be included at the beginning of the source code file.

Conditional Compilation

- Forms:

```
#ifdef macro-name
code to be compiled if macro-name is defined
#else
code to be compiled if macro-name is not defined
#endif
```

or

```
#ifndef macro-name
code to be compiled if macro-name is not defined
#else
code to be compiled if macro-name is defined
#endif
```


Using Conditional Compilation

- Some functions are defined to be used by both C and C++ programs.
- If a C/C++ compiler is compiling a program as a C++ program, then the macro __cplusplus is defined. (Note the two _ chars).
- Then the function would be declared as follows:

```
#ifdef __cplusplus
extern "C" {
#endif
function declaration
#ifdef __cplusplus
}
#endif
```


Preventing Multiple Includes

- A header file may be included by another header file.
- The user of the header file may not know this and may include a duplicate.
- This may lead to a compile error.
- To prevent this, each include file should be structured as follows:

```
#ifndef unique-name
#define unique-name
...
#endif
```

- Generally *unique-name* is related to the file name.
  - Example myfile.h would use the name MYFILE_H_

More on #include directive

- The #include directive has two forms:

  #include <*header*>

  – is reserved for standard library headers.

  #include "*file-name*"

  – is used for user-defined include files.
- The convention is that user-defined include files will end with the extension .h.
- Note that the standard library headers do not end with .h.