C++ Primer Notes

Kat

March 11, 2020

Contents

Chapter 1

Getting Started

1.1 Introduction

Every C++ program has one or more functions, with one of these functions being `main()`. A function is defined with four parts:

R.T.	Return type
F.N.	Function type
P.L.	Parameter list, maybe empty
F.B.	Function body, inside braces

For example, a basic `main()` function would look like this:

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

`int` is the return type, with the function `main` requiring an `int` R.T. A semicolon (`;`) closes a statement inside a function. For `main()`, the function returns a status indicator. Thus `return` is required, with `0` indicating a success.

Every data element (called objects in C++) must have a type. The type lets the compiler know what operations are possible on the object. For instance, say we have a variable `v` and the type of `v` is `T`. It would be described as "`v` has type `T`" or "`v` is a `T`". Types are integral to C++ and must always be given for any object.

1.2 Compiling

C++ is a compiled language, meaning that a compiler is required to take the human friendly language to something a computer can understand. This is in contrast to a language like Python in which the language you write in is the language that is run.

1.2.1 The GNU Compiler Collection: GCC

Since I am using Linux, the primary C++ compiler of use is GCC. While this has other compilers for different languages, we are concerned with `G++`. For simple programs, the primary usage is as follows:

```
$ g++ -o output input.cpp
```

1.3 Input/Output

C++ doesn't natively handle input and output operations but relies on a built in library called `iostream`. C++ gets input/output data via a stream, a sequence of characters read or written to an IO device that is generated or consumed sequentially. In the `iostream` library there are two types of streams: `istream` and `ostream`. There are a handful of IO objects in this library that we can classify:

Function	Use	Note
`cin`	Standard input	Type `istream`
`cout`	Standard output	Type `ostream`
`cerr`	Standard error	For general errors
`clog`	Standard log	For general info on the program

1.4 Namespaces

C++ has many functions, and some share names between libraries. The compiler and author have to know what object one is referring to. To do this, we prepend a namespace to the object in question and link them with a scope operator:

```
std::cout
```

`std` is the standard C++ namespace and most objects in the standard libraries use this namespace. `::` is the scope operator and it lets us describe a namespace within a scope.

1.4.1 Headers

A header links to a library and we use them in C++ programs via the `#include` director. This is used outside of the function and tells the compiler to include the library while compiling. It is used like so:

```
#include <iostream>
```

1.5 Comments

Comments are integral to any programming language. They improve readability and help the author and people reading the code to better understand the code at hand. In C++ there are two kinds of comments: single-line and paired.

1.5.1 Single-line Comments

Single-line comments are made with two forwardslashes, `//`. Everything past this is not read by the compiler up until a newline is made. For example:

```
std::cout;; // this comment keeps the code in view of the compiler
// std::cout;
// in the line above, the code is commented out
```

1.5.2 Paired Comments

A paired comment lets one create large blocks of comments, particularily on multiple lines, without having to use single-line comments for each line. A paired comment is started with `/*` and ends on the *first* instance of `*/`. This last part is important and means we can't nest paired comments. If we wanted to comment out a section of code that contains a set of paired comments, we would be unable to. For instance:

```
/* we start our comment here
stuff /* paired */
we end our paired here */
```

In this example the paired comment ends in the second line at the first `*/`. This leaves the second `*/` without an initial `/*`. This block would thus be invalid. And so in order to comment out paired comments, one should use single-line comments for every line involved.

1.6 The *for* Statement

In C++, `while` loops are very common. The most common of these are `while` loops that increment a value until it reaches a condition set by the author:

```
while ( i < 10 ) {
        do stuff;
        ++i
}
```

Since this `while` loop is so prominent, C++ introduced a new function to replicate it simpler: the `for` loop. A `for` loop contains three parts in its header: a init statement, a condition, and an expression.

Part	Example	Description
Init statement	`int val = 1;`	Defines a variable for the loop
Condition	`val <= 10;`	Describes when to end the loop
Expression	`++val`	What to do after each loop

And all together this would become:

```
for ( int val = 1; val <= 10; ++val) {
        stuff;
        maybe more stuff;
}
```

It is important to note that only the first two parts are ended by a semicolon, the expression is not ended by a semicolon.

1.7 Data Structures

In C++ we often want to be able to define our own objects, types, and functions. This is, in fact, what makes C++ so powerful. We can arbitrarily add in our own classes that behave like standard classes. A data class defines a type along with a collection of operations related to that type. In order to include these we must have a file (typically '*.h') and include it into our program. We can do that with `#include "Our_class.h"`. In a class we can have member functions, functions defined as part of a class. These are sometimes called methods. We use these on objects of the class type and format as such: `object.method()`.

1.8 Notes

- A semicolon ends a statement.
- The scope operator (`::`) is used to define the name of an object.
- We can read from an input stream by using `while ( cin << input )`
- We can read from a file and output to a file:
  `program <infile >outfile`

Chapter 2

Variables and Basic Types

2.1 Arithmetic Types

There are many types in standard C++ libraries. The arithmetic types allow arithmetic operations to be applied to them. There are two basic types: integral and floating point types. Both allow different sizes for different uses and integral types can be either signed or unsigned. It is best practice then to restrict the types one uses. If only integer math is needed then floats shouldn't be used. If a value will never be negative, the integer should be unsigned. Floats should always be double precision, they are more accurate and take up a bit more in memory. Types like char and boolean should never be used for arithmetic.
We can convert one type to another on the fly depending on some conditions. This lets us treat some variables differently when we want to. There are some limitations and conditions we should be aware of:

Conversion	**Conditions**
Non-bool → bool	0 is false, otherwise true
Bool → non-bool	True is 1, false is 0
Float → integral	Truncated, integer component is kept
Integral → float	Fractional part is 0, can lose precision
Out-of-range → unsigned	Remainder of the value modulo
Out-of-range → signed	Undefined, produces unexpected results

2.2 Literals

Literals are self-evident value. For instance "3" is a literal. We know what it means just looking at it. There are many types of literals:

Type	Example
Decimal	3
Octal	0#
Hexidecimal	0x#
Character	‘c’
String	“word”

2.3 Escape Sequences

Sometimes we want to enter a specific kind of input, but due to the way C++ we catn't input them as we would want to. We use a backslash \ folowed by a neccessary character. There are many escape sequences and they can be used in many places:

Escape Sequence	Function
\n	Newline
\t	Horizontal tab
\a	Terminal alert/bell
\v	Vertical tab
\b	Backspace
\“	Double quote
\\	Backslash
\?	Question mark
\‘	Single quote
\r	Carriage return
\f	Form feed

One can also use octal/hex values with escape sequences to insert a specific character (such as a non-English character, symbol, ect...)

2.4 Variables

Variables in C++ are named storage that can be manipulated and has a type. The type determines size and layout of memory, its range of values, and set of operations possible on the variable.

2.4.1 Creating Variables

Creating variables is the first thing a variable needs. There are two main ways to do it: a declaration and a definition. A definition is a *declaration* and an *initialization*.

Declaration

A declaration makes a name known to the program along with its associated type. A declared object does not have any value associated with it, this is known as **default initialization**. For built in types, default initialized values are 0 when called declared outside of a function and undefined when inside a function:

```
int i;   // value is 0
int main() {
        int i;   // value is undefined
}
```

Classes can be supplied with their own default values.

Initialization

Initialization is when you declare and immediately give a value to an object as opposed to assignment which clears out the existing value and gives it a new one. There are 4 main ways to initialize an object:

```
int x = 0;
int x = {0}
int x{0}
int x(0)
```

Initializations with curly braces are known as **list initialization**. These are very specific and the compiler won't compile our code if information is lost (say a float is used in an initialization for an integer)

A definition combines these two things, and as a result you can only define once. You can, however, declare multiple times. This is important if we want to carry one variable from one file to another. To do so, we declare or define the variable in a main file then use the keyword `extern` to only declare the variable in other files.

2.5 Identifiers

We can name objects in many different ways, but there are some rules. Names used by standard libraries are unavailable (e.g. `cin`) for use. Underscores can't be used more than once in a row, can't begin with an underscore followed by an uppercase letter, and identifiers defined outside of a function can't begin with an underscore. Aside from those exceptions, the rules are fairly lax. Any combination of numbers and letters can be used. Some common ways to create an identifier can be:

- `wordName`
- `word_Name`
- `Word_name`

2.6 Scope

A scope is a part of a progam where a name has a certain meaning. Usually these are things enclosed in curly braces. A scope can be nested. A `for` inside a `main()` function is nested. Variables defined in the `main()` function are seen by any nested scopes, but a variable defined in a scope is only seen by further nested scopes. This is overridden by the scope operator (`::`).

2.7 Compound Types

A compound type is a type defined in terms of another. C++ has multiple compound types but the two discussed here are **reference** and **pointer**. Defining these types are a bit more complex than normal types.

2.7.1 References

A reference defines an alternative name for an object. A reference refers to another type. We use `&d` for definition, with d being the name being declared. A reference binds to an initializer of another type, and because of this the object being reference *must* be initialized. The reference we are defining must also be the same type as the object we are referencing. It also can't be a literal. We also can't define a reference to a reference.

2.7.2 Pointers

A pointer points to another type. It allows indirect access to the object that it points to, but unlike references, it's its own object. It doesn't need to be initialized. We define a pointer using `*d` with d being the pointer name. Pointers hold the address of the object it points to. We can get the address using the address-of operator (`&`). The value stored in the pointer can be one of 4 states:

1. Point to an object
2. Point to the location just past the end of the object

3. Null, the pointer points to nothing

4. Invalid, any value not the ones above

In order to access the object in which the pointer points to we use dereference the pointer using the dereference operator (`*`). This allows us to perform most operations on the object without modifying the pointer itself.
Since pointers are objects, we can assign them new values. However, it can be a touch confusing if we want to change the pointer or the object it points to. To change the pointer we *don't* dereference the pointer. Assigning the object a pointer points to requires a dereferenced pointer.
We can also use pointers in conditions. A pointer is false if its equal to 0, and true otherwise. We can compare two pointers as well. They are equal if they hold the same address (not just the same value).
We can also apply multiple type of modifiers to a declarator. This lets us have pointers that point to pointers. References to pointers can also be created, but pointers to references cannot as references are not objects. To create a reference to a pointer we simply use the **dereference operator** while declaring the pointer.

Null Pointers

Null pointers can be useful when we don't have an object to point to but still need the pointer around. The best way to initialize a null pointer is to use `nullptr`. There are other methods, but require a preprocessor variable. It is always best to initialize a pointer to something.

void* Pointers

As pointers are objects they neccessarily have a type. This lets us perform operations on them like any other object. However, that sometimes can get in our way. If we care more about pointers as an address of memory, we don't want to be concerned with the type of the pointer. The `void*` pointer lets us create a pointer of unknown type. We can create it simply by type `void *p = &obj`. That pointer can hold an object of any type and a pointer of any type.

2.8 const Qualifier

There are many cases in which we want to keep a variable unchanged. To do this we simply add the qualifier `const` to the beginning of the declaration

of the variable. When a `const` is created we cannot change the variable and we will cause an error. Uninitialized `const` variables will also cause errors. However, we are allowed to perform any operation on a `const` that doesn't change the variable.

2.8.1 const References

A reference to a `const` cannot be used to change the value. In this case it means we can't make a non-`const` reference to a `const` variable. We can initialize a reference to a `const` with a type that isn't the type of the reference. The compiler does this by creating a temporary variable that holds a `const` of the type in question. A reference to a `const` may refer to a value that isn't a `const`. References to `const` are thus used when we don't want to change the *reference* only.