C++ Primer Notes

Kat

April 27, 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.

2.8.2 const Pointers

A pointer to a `const` cannot change the object to which the pointer points. However, a `const` pointer can point to a non-`const` object. We just can't change the object to which the pointer points.

2.8.3 Top-Level and Low-Level const

Since a pointer can either be a `const` or point to a `const`, it is important to distinguish what kind of `const` we are referring to. A **top-level const** indicates that the pointer itself is a `const`. A **low-level const** indicates that the pointer points to a `const` object. More generally, a low-level `const` refers to the base of a compound type like pointer or reference while top-level `const` refers to the object itself. This is important when we copy values. Top-level `const` is ignored when copying. This makes sense since copying an object doesn't change the object. On the other hand, low-level `const` *isn't* ignored when copying. The object itself is a `const` and that can't change.
In general, we can convert non-`const` to `const` but not the other way around.

2.8.4 Constant Expression

Sometimes we want the output of a specific expression, but we don't want that value to be changed. We can use a constant expression that is evaluated at compile time that gives us a constant value of an expression. It lets us confirm that what we have is a `const`. Our use of `constexpr` is limited however. It is generally limited to literal types like arithmetic, pointers, and references. `constexpr` pointers apply to the pointer and not the object it points to.

2.9 Dealing with Types

As programs get more complex, types get more complex. Thus we would want to simplify our usage of types as best as we can.

2.9.1 Type Alias

One thing we can do is create type aliases. These are names that are synonyms of another type. One way to create a type alias is by using the `typedef` keyword when declaring a type (e.g. `typedef double wages;`, wages is an alias for double). A limitation though, is that we can only decalare with the `typedef` keyword. An **alias declaration** can simplify this. The syntax is as follows: `using name = type;`. The name is whatever alias name we want. This alias can then be used wherever a type name might appear. An important thing to remember is that an alias made through `typedef` uses the base type and isn't just a name substitution.

2.9.2 auto Type Specifier

When types can get too complex, we can use the type specifier `auto` to have the compiler guess what the type is. The compiler has some quirks with respect to `auto` however. It uses the base object, so reference types are ignored for what the reference refers to. It ignores all top level `const`s so we must specify if the type is a `const` if we care about it.

2.9.3 decltype Type Specifier

When we want to be more specific and get the deduced type of the item in question, we use `decltype`. This gives us the type of the *operand* rather than the object. This returns top-level `const` and the type of the reference, giving us a good bit more specificity.

2.10 Defining a Data Structure

Creating our own data structures is incredibly useful and is relatively simple:

```
struct name {
        x
};
```

With x being our data elements. We prepend this to our program and the program pretends that the data was defined in our program. The names in the class must be unique to the class, and we are able to initialize any objects to give them a default initialization value. We can also make it a separate file and include it in our program. The preprocessor does the same prepending as we did. To do that we use: `#include "Data_structure.h"` with "Data_structure" being our data file and in the same location as the source code is compiled in.

2.10.1 Header Guards

Obviously we want to include this file in every part of the program that needs it. However, we don't want to risk including it more than one time. Doing so will cause issues. To do that we use preprocessor variables to describe the condition of header file. These are:

Variable Type	Variable
Defined	`#DEFINE`
Not defined	No variable
If defined	`#IFDEF` + `#ENDIF`
If not defined	`#IFNDEF` + `#ENDIF`

These should always be written in a way that the header file is included only once if we want it. The preprocessor variables should also be written in caps to avoid any confusion.

2.11 My Notes

- Types are integral to C++
- Types define operations on objects
- Can use header files to define our own types
- Always include header files once using preprocessor variables

Chapter 3

Strings, Vectors, and Arrays

3.1 Namespace Declaration

Previously, we had to specify the namespace of every function/operator we wanted to use. We can instead set the namespace of a function inside a scope by calling the namespace before a function starts. Generally, these are added underneath our included libraries. They are declared using the `using` declarator, e.g. `using std::cout;`. We don't want to use this inside header files however.

3.2 Library string Type

Using the `std` namespace, the `string` library lets us create and modify strings. We must, of course, initialize the string and there are many ways to do so:

Initialization	**Result**
`string s1;`	Default initialization, empty string
`string s2(s1);`	s2 is a copy of s1
`string s2 = s1;`	Same as above
`string s3("value");`	s3 is a copy of string literal, not including the null
`string s3 = "value";`	Same as above
`string s4(n, 'c');`	Initialize s4 with `n` copies of character '`c`'

3.2.1 Direct vs Copy Initialization

A copy initialization uses `=` to copy the right object to the left object during creation. A direct initialization doesn't have this. We use copy initialization for single initialization, but if we want to initialize a variable from more than one value we must use direct initialization (see the initialization of s4).

3.3 string Operators

There are many operators for the `string` type. Here are some examples:

Operator	Function
`os << s`	Writes s onto output stream os, returns os
`is >> s`	Read whitespace separated string from is into s, returns is
`getline(is, s)`	Read line input from is to s, returns is using newline
`s.empty()`	Returns true if s is empty, else false
`s.size()`	Returns the number of characters in s
`s[n]`	Returns reference to character at position `n` in s, starts from 0
`s1 + s2`	Returns string cocantenation of s1 and s2
`s1 = s2`	Replaces s1 with a copy of s2
`==, !=, <, <=, >, >=`	Equalities are case sensitive and use dictionary sorting.

We can read a string from an input buffer in the same manner as we can an integer. The `size()` operator returns the type `string::size_type`. The auto/decltype type specifier should be used when using this operator for convenience. It is unsigned. The cocantenation operator requires one string to work.

In addition to these, there are many functions in the `cctype` header that allow control over the characters of a string.

3.3.1 Ranged for

Modifying characters in a string is a pretty common thing, and in order to do so we use a reference to the characters in the string. Modifying the reference then switching to the next character reference is the best way to work through each character in the string. A ranged for is set up as so:

```
for ( &c : s)
        do stuff
```

3.3.2 The Subscript Operator

If we want to access a single character in a string, we use the subscript operator (`[]`). Starting at 0, the subscript operator uses the `string::size_type`

type as the index/subscript. If the index doesn't correspond to a valid character, the subscript is invalid.

3.4 Library vector Type

A vector in C++ is a collection of objects of the same type, also called a container. Vectors are contained in the vector header in the standard namespace, std. This lets us create *class templates*. A class template is a way for the compiler to generate classes and functions. For vectors, we need to specify that we want to create a vector of a certain type: vector<type> name. This creates the vector type from a template, creating a new type called vector<type>. While vectors of references are not possible, there are many ways to create vectors:

Initializer	**Result**
vector<T> v1	Vector v1 holds object of type T, default initialization
vector<T> v2(v1)	v2 has copy of each element in v1
vector<T> v2 = v1	Same as above
vector<T> v3(n, val)	v3 has n elements, each with value val
vector<T> v4(n)	v4 has n elements of default-initialization
vector<T> v5{a,b,c...}	v5 has number of elements as initializors
vector<T> v5 = {a,b,c...}	Same as above

There's an important distinction that vectors have. Parentheses () are for constructing objects while braces {} are for list initialization. List initialization adds the value to the vector, but object construction creates objects based on specific parameters. The vector vector<int> v1{10} creates a vector with *1* object of value *10*. The vector vector<int> v2(10) creates a vector with *10* objects of value *0*. This is an important distinction and should be thought of.

3.5 vector Operations

There are many vector operations and some are shared between similar library types:

Operation	Function
`v.empty()`	Returns true if v is empty, otherwise false
`v.size()`	Returns size of vector, the number of elements
`v.push_back(t)`	Adds element of value t to the end of v
`v[n]`	Returns reference to element at position `n`
`v1 = v2`	Replace the elements of v1 with the copy of elements in v2
`v1 = {a,b,c...}`	Replace elements in v1 with copy elements in the list
`==, !=, <, <=, >, >=`	Vectors are equal if they have the same amount of elements and each element corresponds to the elements in the other vector, dictionary sorting

We can't use a ranged for loop if the body of the loop adds elements to the vector. We can however, access elements in a vector in the same manner as a string. The size operation returns a `size_type` type. Use of auto/decltype is encouraged.

3.6 Iterators

Iterators allow us to indirectly access elements in a container/object, they let us walk through them. Types with iterators have members that return iterators. We can access the beginning and the end iterators in a container by using the `begin()` and the `end()` operators respectively. `begin()` returns the iteraotr that denotes the first element, `end()` returns the iterator positioned one past the end. These operators don't care about type, and are mostly used to determine if we have processed all the elements.

3.6.1 Iterator Operations

There are a good number of iterator operations:

Operation	Function
`*iter`	Returns reference to element denoted by the iterator iter
`iter -> mem`	Dereference iter and fetch member mem from underlying element, replaces `(*iter).mem`
`++iter`	Increment iterator
`--iter`	Decrement iterator
`==, !=`	Only equalities, same if denote same element or off the end iterators for the same container

3.6.2 Iterators and Types

Library types with iterators define the types `iterator` and `const_iterator`. It is best to use the `const` type when we only need to read from a container. The operators `cbegin()` and `cend()` are constant replacers for `begin()` and `end()` respectively. We can see if a dereferenced element is empty: `(*iter)empty()` or `iter -> empty()`. The parentheses are important in this case because it calls what the iterator references and not the iterator itself.

3.6.3 Iterator Arithmetic

Arithmetic on iterators themselves is very powerful, and some library types support it such as `vector` and `string`. It is important to remember that not all iterators support arithmetic so the following are for `vector` and `string` more generally:

Operation	Function
`iter + n, iter - n`	Add/subtract integral value to/from an iterator yields iterator n places from iterator
`iter += n, iter -= n`	Compound assignment of value `n`
`iter1 - iter2`	Finds the difference in iterator values, produces `difference_type` which is signed
`>, >=, <, <=`	Relational operators, compares value of iterator

Binary Search

A common use of iterator arithmetic is for a `binary search`, used to find a particular value in a container. The set up of a search is as follows:

1. Look at item closest to middle
2. If is desired value, end search
3. If value is smaller than the middle value, search first half
4. Else search second half
5. Obtain new midpoint
6. Repeat until search is complete

3.7 Arrays

An array is a data structure much like a vector, but it is of fixed size and cannot add data element once created. We can create an array like so: `a[d]` with `a` being the array name and `d` being the array dimensions. The number of elements in an array must be greater than 0 *and* the dimension must be a *constant expression*. Arrays, being objects must have a type, either explicitly declared or deduced from a list initialization. Size can be autodeduced from a list initialization by ignoring the element in `[]`. The list cannot exceed the dimensions of the array.

This last part is important, since string literals will add a null character when initializing an array with string literals without list initialization. The sting literal "test" is 5 characters long unless initialized in a list. We cannot initialize an array with a copy of another array nor can we assign one array to another.

3.7.1 Pointers and References

Understanding arrays with respect to pointers and references can get complicated:

Array Definition	Type
`int *ptrs[10];`	ptrs is an array of 10 pointers to int
`int &refs[10];`	ERROR: cannot create an array of references
`int (*Parr)[10] = &arr;`	Parr points to an array of 10 ints
`int (&arrRef)[10] = arr;`	arrRef refers to an array of 10 ints

Defining arrays can get a touch confusing, so its best to work outwards from the array name instead. The first array is a pointer going left, an array of size 10, and those data elements are of type int going left again. This should help with the confusion somewhat. We can't create an array of references since references aren't objects but we can create an array that refers to another.

3.7.2 Accesssing Element in an Array

Accessing elements in an array is very similar to accessing elements in a string or vector. Indices start at 0, and we can use the `subscript []` operator to access specific elements in that container. We can even use a variable for the subscript index, but that must have the type `size_t`.

3.7.3 Pointers and Arrays

Arrays and pointers are intertwined fairly tightly in C++, to the point where there is a special operation in C++ in which we can directly get a pointer to the first element in an array without dereferencing the array exactly:

`int *p = arr;`

The address-of operator (`&`) when applied to an array lets us get a pointer to an object in an array. In effect, pointers to arrays act like iterators in the `string` and `vector` container types. In most operations on objects of an array type, we can always get a pointer to the first element of an array and perform operations on that.

Pointers are Iterators

Since pointers to arrays are so special, we can perform all of the same operations on iterators in the `vector` and `string` container type on pointers to arrays. To effectively use pointers to arrays as interators we need get the first and one-past-the-last elements of the array. Unlike `string` and `vector` types, we can use the subscript operator to get the element one past the last element. Obtaining this element is very similar to the `end()` operator in the `string` and `vector` container type. And while we can obtain the one-past-the-last element in an array in this manner, it requires knowing the array size in some manner and can cause errors.

Fortunately, *C++11* introduces two new functions for arrays: `begin` and `end`. Like the iterator counterparts, these return the first and one-past-the-last elements of the array in question. Unlike them, the array versions are not member functions and thus requirement an argument, one array in this case:

```
int *beg = begin(arr);
int *last = end(arr);
```

These functions are defined in the `iterator` header.

Pointer Arithmetic

Pointers to arrays can take all of the same iterator arithmetic as outlined previously. When taking the difference between two pointers, the result is of type `ptrdiff_t` in the header `cstdef`. Relational operators on pointers to an array only work when using related objects. As we are dealing with pointers, dealing with types can be a touch confusing. Just remember what you mean with respect to dereferencing pointers to arrays.

3.8 Interfacing with Older Code

C++ in many programs will need to interface with other coding languages, namely C. It is common to come across this older code and C++ does allow native use of this older style. There are two prominent examples here:

3.8.1 C-Style Character Strings and Library string

C-style character strings are strings initialized with a series of characters followed by a null character. They are, in function, an array of characters. We can use C-style strings when we want to use library `string`s but we can't do the reverse. Luckily, the `string` library has a member function that lets us return a C-style string: `c_str` We give it an established library `string` and it returns a C-style string with the type `const char*`. As these strings are arrays, we can't add to them and we have to manage the pointer aspect of arrays. If we have a pointer to a C-style string the pointer must point to a null terminated string. C-style character strings are initialized like so: `char ca[] = "example";`, giving us a character string "example" which has 8 characters in total.

Operation	Function
`strlen(p)`	Returns length of p not including the null character
`strcmp(p1,p2)`	Compares p1 and p2 for equality; returns 0 if p1 == p2, 0 > if p1 > p2, & 0 < if p1 < p2
`strcat(p1,p2)`	Appends p2 to p1, returns p1
`strcpy(p1,p2)`	Copies p2 into p1, returns p1

We cannot compare C-style strings since that would be an operation on pointers, we must use `strcmp`. We also cannot directly cocantenate and copy C-style strings into one another, `strcat` and `strcpy` must be used respectively. The trouble is that an appropriately sized C-style string must be used to pass off these values and can cause errors since the size of the C-style string *must* be an appopriate size. This is a huge source of errors, bugs, and security problems.

3.8.2 Initializing a vector with an array

We can't initialize an array with a vector nor can we initialize an array with an array. We can, however, initialize a vector with an array:

```
int arr[] = {0,1,2,3,4,5,6,7,8};
vector<int> ivec(begin(arr), end(arr));
```

In effect we are giving the vector the range of some values in an array. It doesn't have to be the first and one past the last value of the array. We can use any value of the array subscripted:

```
vector<int> subVec (arr + 1, arr + 4);
```

which gives us a vector filled with values from the array starting at arr[1] and going to arr[3]. The array values used aren't iterative, so we start from arr[0] then add 1, start from arr[0] then add 4 giving use arr[1], arr[3].

3.9 My Notes

- Don't use arrays and pointers unless really needed, this includes C-style character strings

Chapter 4

Expressions

An expression contains one or more operands and yields a result when evaluated. The most simple expression is a literal. When evaluated they are the value of the literal. There are two major type of operators: **unary** and **binary**.

- Unary operators act on one operand, such as the address-of (`&`) and the dereference (`*`) operator

- Binary operators act on two operators, such as the equality (`==`) operator and the multiplication (`*`) operator

There are more operator types but aren't very common.

4.1 Understanding Expressions

Understanding expressions requires knowing the precedence and associativity of the operators and the order of evaluation of the operands.

Operands of different types can be converted to a common type in an expression if the types in question are similar. For instance, an `int` and `float` can be converted to a common type but a `string` and an `int` cannot. Small integral types (`bool`, `char`, `short`) can be converted to a larger integral type such as `int`.

We say that an operated is *overloaded* when it has multiple meanings depending on the class type in question. The IO library `>>` and `<<` operators are also used in the `string` and `vector` libraries and with iterators.

4.2 lvalue and rvalue

Since C++ inherits a lot from C, it takes the idea of `lvalues` and `rvalues`. Every expression in C++ is either a `lvalue` or a `rvalue`. In C, `lvalues` stood on the left-hand side of assignment whereas `rvalues` did not. In C++ they have a far more robust meaning. Generally a `lvalue` resturns an object or a function while a `rvalue` returns the objects content. Essentially `lvalues` refer to the place in memory of an object (much like pointers). Because of this we can use an `lvalue` when an `rvalue` is needed but not the reverse.

Many expressions are `lvalues`:

- Assignment: uses a non-`const` `lvalue` as left-hand operand and yields it left-hand operand as an `lvalue`.
- Address-of: `lvalue` operand, returns a pointer to a operand as an `rvalue`.
- Built-in, `string`, and `vector` dereference and subscript operators all yield `lvalues`.
- Built-in iterator increment/decrement use `lvalue` operands, the prefix versions yield `lvalues` as well.

4.3 Order of Evaluation

While precedence grouping is useful, it lets us know what operations evaluate before others, it doesn't say everything. Precedence in no way tells us in what order operators of the same group evaluate. This causes a ton of issues in programs, as the compiler may not detect or do what we want it to. As a result, unless order of evaluation is specified, it is best practice to never refer to and change an object in an expression if there is no orderc There are four expressions in which order of evaluation is clearly defined:

- `AND (&&)`
- `OR (||)`
- `Conditional (? :)`
- `Comma (,)`