Ohlone College
CS 124-03 Programming with Data Structures Using C++

Notes and Examples: C++ Basics

What is a C++ program?

A C++ program is *essentially* a collection of *functions*. C++ functions, like their counterparts in other programming languages (variously referred to as functions, *procedures*, *subroutines*, *methods*, and so on), encapsulate the code necessary to perform a single task. Functions are *called* to perform their task, and then *return* a result when they are finished. Each function has a name, accepts a set of *parameters*, does a task when called, and returns a value (or, perhaps, no value at all).

Executing a C++ program is generally done by calling a function with the name **main**. The C++ Standard requires **main** to return an integer value, though it does not need to take parameters. (The **main** function can take parameters, as you will see later in the course), The program begins with the call to **main** and ends when **main** returns.

One of the most basic C++ programs that you could write is shown in the example below, which contains only a **main** function that does nothing except return the integer value 0 back to the operating system.

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

There are a few things worth noting about this simplest of examples.

The first line of the function is called its *signature*. The signature specifies how to call the function, consisting of three parts:

- A *return type* (i.e., what type of value the function will return). In this case, it is specified that the main function will return an integer value, which is represented using the built-in C++ type **int**.
- A *name*. In this case, that name is **main**.
- A sequence of *parameters* surrounded by parentheses. In this case, there are no parameters, yet it is still necessary to write the parentheses.

After the function's signature, is the function's *body*, which specifies what the function actually does when it is called. The body is written as a sequence of *statements*, surrounded by characters { } often called *curly braces*. The only task that this example **main()** function does is return the value 0.

Unlike in some languages, there are few specific layout rules (e.g., spacing, indention). For example, this same function could have all been written all on a single line:

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

However, as explained in our course C++ coding guidelines, use spacing and indention to highlight the syntactic structure of functions. In this course you will be expected to follow Coding Style guidelines and for good reason: Purposefully structured code is easier to read and understand (particularly for others who did not write it, or even for the author months or years later after the details have been forgotten).

Where does main()'s return value go?

Ordinarily, C++ functions are called by other C++ functions. As in many programming languages, when the called function returns a value, the calling function can then make use of that value.

But this brings up an interesting question: If the **main()** function is the entry point of the entire program, why does it return a value, and to whom is the value returned? Every program running on operating systems like Windows, Mac OS X, and Linux returns an *exit code* back to the operating system when it is done. This makes it possible to coordinate the execution of many programs (e.g., using *shell scripts* on Linux) and have that coordination be done differently depending on the result of each program. Most commonly, an exit code of 0 is used to indicate *success*, while other exit codes are used to indicate various kinds of failures; checking for an exit code of 0 from the first program might indicate which program should be run next, or whether to run a second one at all. This kind of thing is out of the scope of our work in this course, so almost always return 0 will be the closing statement from **main()**.

Compilation, types, and static type checking

When you use the C++ programming language there is a stepwise process you follow each iteration of your design *build.*

- *Edit your code*, making whatever changes you wanted to make (e.g., adding a new feature, or fixing a previous mistake).
- *Compile your program*. In short, *compilation* is the process of taking the code you wrote and translating it into an executable program (i.e., machine code capable of executing directly on

your chosen target machine). In the process of performing that translation, the compiler checks for a broader range of mistakes, and compilation fails if your program is incorrect either syntactically *or semantically*. If compilation fails, you go back to the first step and edit your code again.
- *Run your program*. Once compilation has succeeded, you can then try to run your program. On the other hand, if run time execution fails, you will be *unable* to complete the successful run of your program, which means that certain kinds of run time errors (e.g. attempted division by zero) need to be corrected. Once again, you go back to the first step and edit your code.

One of the key ways that C++ differs from a language such as Python, is that C++ takes the idea of *types* — what kind of values are permitted to be stored in a variable, returned from a function, and so on — very strictly.

When you want to use a variable or a parameter, or when you want to return a value from a function, you will need to specify what type of value you intend to store, pass, or return.

- The compiler will check that the values you store in a variable, pass into a parameter, or return from a function match the intentions stated in your code. When you try to, say, return a string from a function that you have said should return an integer, the compiler will emit an error message, *and compilation will fail*.

Overall, this process is called *static type checking*. The word *static*, in this context, can be taken to mean "Done before the program runs" (i.e., by the compiler).

A slightly more full-featured example

Let's add some code to our **main()** function so that it has a visible effect when we run it. In particular, let's write a program that writes a string to the standard output (so, for example, if we run the program from a shell prompt in Linux, you will see the output in the shell).

```
#include <iostream>


int main()
{
    std::cout << "Hello CS 124!" << std::endl;
    return 0;
}
```

There are a few notable things going on in this example:

- The line beginning with **#include** is called a *preprocessor directive*. Preprocessor directives begin with # characters. For now, you can think of the **#include** directive as loading a library; in this case, the library being loading is part of the C++ Standard Library (you can tell that it is not part of your code because its name is surrounded by angle brackets < >) and its name is **iostream**, which gives us the ability to manipulate *streams of input or output*.
- Items that are defined in the C++ Standard Library generally have names that begin with **std::**. In this case, **std** is what is called a *namespace*. In this course, when we use items that are defined in namespaces, we will not necessarily use their full names (including the names of their namespaces). Do be advised that by using an explicitly stated namespace allows you to more easily distinguish between items that are standard and items that you authored. (This kind of careful differentiation can become much more important as programs become larger, even if it seems overkill in short, simple examples like these.)
- **std::cout** is an object called an *output stream*; in particular, it is the one that represents the program's *standard output*. Whatever gets written into it will be written to the program's standard output channel. If you run a program from a shell prompt in Linux, for example, then you would see all of that output appear in the shell. You can *put* data into an output stream by using the **<<** operator (which is often called the "put operator"). Note, too, that we can string together several uses of **<<**, so that we can easily write a sequence of items into the same output stream in a single expression.
- **"Hello CS 124!"** is a *string literal*, i.e., a literal sequence of characters of text. String literals follow similar rules in C++ as they do in many other programming languages, though one thing to note in C++ is that they must always be surrounded by double-quotes, as opposed to being surrounded by single-quotes (e.g. an option in Python).
- **std::endl** is an object that represents a platform-independent end-of-line sequence. For our work, it is a good idea to use this instead of something like **"\n"**, since end-of-line sequences are different on different operating systems.

With those things in mind, we can see that executing this program would cause **Hello CS 124!** to be written to the program's standard output, and then the program's exit code would be 0. For example, in Linux you can check the return code of your program run as shown below:

```
$ ./a.out
Hello CS 124!
$ echo $?
0
```

Basic built-in data types

C++ has a set of basic, built-in data types that are available in all programs by default. They are not part of a library, but rather are intrinsic to the language. A non-exhaustive list of those built-in types follows.

- *Integral types*, which represent integer values. There are multiple integral types in C++, not just one, differing based on the basis of *how much memory* is used to represent a value of each type. The common examples are: **int**, **long**, **short**, and **char**. The size of a value of each type is specific to the platform you are using (e.g., the operating system, the compiler, and so

on), as opposed to being standard. The only thing you can truly count on from one platform to another is the following rule:

- **sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)**

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

int main()
{
    cout << "char size: " << sizeof(char) << " byte." << endl;
    cout << "short int size: " << sizeof(short int) << " bytes." << endl;
    cout << "int size: " << sizeof(int) << " bytes." << endl;
    cout << "long int size: " << sizeof(long int) << " bytes." << endl;
    return 0;
}
/*
char size: 1  byte.
short int size: 2 bytes.
int size: 4 bytes.
long int size: 8 bytes.
*/
```

- *Unsigned integral types*, which represent whole number values (i.e., they cannot be negative). Each of the integral types has an unsigned variant, denoted by the word **unsigned** appearing in the name of its type.
- **bool**, a boolean type that has two possible values: **true** and **false**. It turns out that **bool** values are freely exchangeable with integral values, for the most part, with 0 typically meaning **false** and anything non-zero meaning **true**.
- *Floating-point numbers*, which represent numeric values that can have a fractional part. There are two such types built into C++: **float** and **double**, with the distinction being how many bits of *precision* they offer — more or less, how many significant digits are available, though the digits are binary as opposed to decimal. The number of bits is different depending on the platform, but **sizeof(float) ≤ sizeof(double)**.

```
#include <iostream>
```

```
using namespace std;


int main()
{
    cout << "float size: " << sizeof(float) << " bytes." << endl;
    cout << "double size: " << sizeof(double ) << " bytes." << endl;
    return 0;
}
/*
float size: 4 bytes.
double size: 8 bytes.
*/
```

Expressions and statements

Function bodies are made up of *statements*, which can, in turn, contain *expressions*. While you may have written programs in a language that draws a distinction between expressions and statements, you not actually had this distinction called out before.

An *expression* evaluates to a value, meaning that it also has a type. Expressions are quite often built by combining smaller expressions. The type of an expression is determined — at compile time — based on the types of its subexpressions and the operators or other syntactic constructs used to combine them. When the program runs, the value of the expression is determined by combining the values of the subexpressions.

An example of an expression with subexpressions would be **a** + **b**, which appears to be the addition of the variables **a** and **b**. **a** and **b** are themselves subexpressions, and their types are governed by the types of the corresponding variables. They are combined using the operator +, a symbol whose meaning actually depends on the types of **a** and **b**. For example, if **a** and **b** are integers, this means that we want to add the two integers together and the value of the expression is their sum; if **a** and **b** are strings instead, then we want to concatenate the two strings together and the value of the expression is their concatenation. Either way, since the compiler will always definitively know the types of **a** and **b**, the expression will have a definitive meaning before the program runs.

Statements perform a job, but do not themselves have a value (or a type). Their primary role is to have a *side effect*, which is to say that executing them causes your program *state* to change. The value of a variable may be changed, control flow in the program may be affected, input may be read from an input source, output may be written to an output source, a file may be created, a socket may be opened, etc. Statements quite often contain expressions within them, and the values and types of those expressions affect what they do.

Expression statements, compound statements, and control structures

C++ provides the ability to write many kinds of statements. The simplest kind of statement is an *expression statement*, which is simply an expression terminated with a semicolon:

a + b;

An expression statement simply evaluates its expression and then discards the value afterward. That may seem useless, but it can be useful when an expression has a *side effect*, such as the assignment of a value into a variable:

a = 3;

Assignment is an expression in C++.

A *compound statement*, also called a *block statement* or a *block*, is a sequence of statements surrounded by curly braces { } (also referred to as a logical code block). When a compound statement executes, the statements inside of it are executed in order, one after another. Compound statements can appear anywhere, though you most often use them as the body of *control structures*.

Control structures are statements that affect the *control flow* of a program (i.e., to affect what happens next, as opposed to the default behavior of one statement flowing straight through to the one after it). C++ has several control structures, many of which are likely to look familiar to you from whichever language(s) you have learned previously.

- **if** statements allow you to execute statements conditionally, based on the value of a *conditional expression*.

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

int main()
{
    int num1, num2;

    cout << "Enter num1: ";
    cin >> num1;
    cout << "Enter num2: ";
    cin >> num2;
    if (num1 < 2 && num2 > 4)
    {
        cout << "out of range" << endl;
    }
```

```
    else
    {
        cout << "in range!" << endl;
    }
    return 0;
}
/*
Enter num1: 1
Enter num2: 5
out of range
*/
```

In this example, **out of range** *or* **in range!** will be displayed to the standard output, depending on whether it is true *or* false that **num1** is less than 2 and **num2** is greater than 4. The conditional expression is the one written in parentheses just after the **if**, and there are a couple of things you should know about it that might surprise you:

- The parentheses are part of the syntax, so they are not optional in this case. The conditional expression must appear in parentheses.
- The body of an **if** statement and any **else if** or **else** branches is made up of a single statement, though that single statement can be a block statement (as has been used here). I suggest that you generally always use blocks, as this practice aids in readability and maintainability.
- Quite often conditional expressions are written that are boolean in nature, they can technically be anything integral, where zero is treated as false and *anything non-zero* is treated as true.

switch statements provide another, more limited form of conditionality, where you execute one sequence of statements or another based on the value of a *controlling expression*.

Example:

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

int main()
{
    int select;

    cout << "Select a number (0..1): ";
```

```
    cin >> select;

    switch (select)
    {
       case 0:  cout << "zero" << endl;
                break;
       case 1:  cout << "one" << endl;
                break;
       default: cout << "not an option" << endl;
                break;
    }
    return 0;
}
/*
Select a number (0..1): 1
one
*/
```

As with the conditional expression in an **if** statement, the controlling expression here must appear in parentheses, and it must be integral. The actual meaning of this **switch** statement is not quite as it appears:

- First, the controlling expression is evaluated. In this case, let's suppose that the value of **select** is 1.
- Based on that value, the corresponding **case** is chosen. The **default** case, if specified, is chosen when no other **case** matches.
- Once a case is chosen, the sequence of statements below it is executed until one of two things happens: the end of the **switch** statement is reached, *or* a **break** statement is reached.

(*Note*: Forgetting the **break** means that you would continue executing statements that comprise the next case. For example, if there were no **break** statements in the **switch** statement above, and if **select'**s value was 1, you would see both **one** and **not an option** appear in the standard output.)

There are a few kinds of loops in C++. Let's start with three of them, each of which is based around a *conditional expression* that follows the same rules as the **if** statement (i.e., non-zero is true, zero is false), and each of which determines whether to continue executing based on whether that conditional expression is true. The three loop variants are shown below.

```
#include <iostream>
```

```
using namespace std;

int main()
{
    int counter = 3;
    while (counter < 5)
    {
        cout << counter << endl;
        counter++;
    }

    counter = 3;
    do
    {
        cout << counter << endl;
        counter++;
    }
    while (counter < 5);

    for (counter = 3; counter < 5; counter++)
    {
        cout << counter << endl;
    }
    return 0;
}
/*
3
4
3
4
3
4
*/
```

These three kinds of loops are similar, but subtly different, so it is worth understanding the details.

- In a **while** loop, the conditional expression is evaluated before each loop iteration (including the first), and stops executing when the conditional expression evaluates to false.
- In a **do..while** loop, the conditional expression is evaluated *after* each loop iteration, and stops executing when the conditional expression evaluates to false. Note that this means that the body of a **do..while** loop will always execute at least once, which can make it useful in scenarios like reading user input with error checking, when you know at least one attempt will need to be made.
- The last of the three is the **for** loop, which is actually a simplification of a common pattern for writing **while** loops. What lies between the parentheses on the top line of a **for** loop consists of three parts, separated by semicolons: the *initialization expression*, the *conditional expression*, and the *iteration expression*. When the **for** loop begins, its initialization expression is evaluated once. Then the loop begins; like a **while** loop, the conditional expression is evaluated before each loop iteration. The body of the loop is executed in each loop iteration, and after each one, the iteration expression is evaluated once. Note that the **for** loop above and the **while** loop above are equivalent to one another.

Declarations

A *declaration* introduces a name into a C++ program. The declaration associates that name with a type, which specifies how it can be used properly, and which gives the compiler the ability to check those uses at compile time to ensure that they're proper. The goal of a declaration is to state that something exists and to give it a type, though it does not necessarily provide all of the other relevant details.

A few examples of declarations follow:

```
int count;

double a_double = 3.0;

int square(int num);
```

The first of these is the declaration of a *variable*. Variables name a block of memory in which a value can be stored. Note that variable declarations include not only a name, but also a type; for example, given the declaration of the variable **count** as an **int**, the compiler will ensure that only an integer value will ever be stored in **count.**

The second of these is also the declaration of a variable, but it also initializes the variable's value at the same time. (One important thing to note about variables: they are not initialized to anything in particular, unless, you assign a value into them, so after the declaration of **count** above, the value of **count** will be undefined, in the sense that there are no guarantees about what that value will be. A subsequent use of that variable before assigning a value to it would be what is called *undefined behavior*, which means that you cannot be sure what will happen next.

The third is the declaration of a *function*. Note that it looks just like the signature of a function, yet it is terminated with a semicolon instead of being followed by a body. The declaration of a function establishes the function's existence, but does not specify what the function does.

Why do we need declarations?

Declarations establish how a name can be used legally in a C++ program. Since C++ performs static type checking, it needs to be aware of what the appropriate types will be; declarations serve this purpose. Names in C++ must be declared *before use* (i.e., the declaration of a name must appear in a program before the first use of that name). This is a bit different from many other programming languages, which cast a much less scrutinizing eye on the order in which you say things.

As a result of the rule requiring declaration before use, the function as written below would be illegal because the variable **a** is not declared until after it has been used.

```
int main()
{
   apple = 2;
   cout << apple << endl;
   int apple;
   return 0;
}
```

Similarly, this function would be illegal, as well because apple has not been declared at all.

```
Int main()
{
    apple = 3;
    cout << apple << endl;
    return 0;
}
```

Definitions

A *definition* gives a declared name a unique, specific meaning. For example, the following code is a definition of a function **prize()**, because it not only specifies that the function exists, but also specifies what the function does.

```
int prize()
{
    return 50
}
```

Since the example above includes the function's signature, this definition would also serve as a declaration if the name **prize** had not yet been declared.

Some declarations are also definitions. Variable declarations not only specify that a variable exists and give it a type, but they also cause memory to be allocated in which to store the variable's value. In a sense, all there is to know about a variable, in order to qualify it is the type, because the type dictates the amount of memory required to store it. Note that variables can be initialized to a value when declared, as seen before, but a variable declaration is also a definition whether you initialize its value or not; variables live even when they have not had a value explicitly assigned to them.

Not all declarations are definitions. A notable example is a function declaration, which specifies how a function can be called, without stating anything about what the function does. (It should be noted that function declarations serve a bigger purpose than you might think, not only because names must be declared before use, but especially when we soon start writing programs that consist of more than one source file.)

Assignments, lvalues, and rvalues

One very fundamental task in C++, as in many other programming languages, is the ability to store a value in a variable. As seen above, *assignment* is a kind of expression that does just that. The expression **a = 3** indicates that the value **3** should be stored into the variable **a**, subject to successful static type checking (i.e., **a** must be able to store an integral value, so it must be a type such as **int**). In this case, **a** and **3** are subexpressions, and the = operator is used to connect the two subexpressions together.

Embedded within this seemingly simple assignment is an important concept in C++, which is the distinction between *lvalues* and *rvalues*. Expressions can be thought of as evaluating to either an lvalue or an rvalue, and it is important to understand the difference.

Originally, the terms "lvalue" and "rvalue" took their names from the distinction between what is legal to put on the ***l***eft- or ***r***ight-hand side of an assignment expression, where an "lvalue" refers to an expression that would be legal on the left-hand side, while an "rvalue" refers to an expression that would *only* be legal on the right-hand side. The following examples embody the concept at work here. (Let's assume that the variables **a**, **b**, and **c** that appear below all have the type **int**.)

```
a = 3     // legal
a = 3 + 4 // legal
a = b     // legal
```

```
a = b + c // legal
3 = a     // not legal
b + c = a // not legal
```

While there is more nuance to these terms than is seen here, the assignments above provide the basic mental framework for understanding them. An lvalue is a value that lives beyond the expression in which it is originally evaluated; variables certainly fit into that category, since their values can be used again later. (A simple way to think about that is to think about whether an expression can be used to represent a location in memory; if so, it is an lvalue.) An rvalue, on the other hand, is one that is temporary, i.e., that dies when the expression that generated it is finished being evaluated. That is certainly true of the value of a constant, or the value of expressions like **3 + 4** or **b + c**, which calculate a result, but do not themselves store it anywhere.

C++ goes to the trouble of naming these concepts partly because they are not as simple as they sound. As you will see, not all lvalues are variable names. For example, the result of a function call can be a location in memory, meaning that it can be used as an lvalue, leading to the following perhaps nonintuitive but perfectly legal line of code:

v.at(2) = 3;

Why we need to take the time to learn this distinction now is because terms like *lvalue* and *rvalue* show up in error messages that emanate from C++ compilers, and because they foreshadow concepts that will be very important from a performance perspective as we consider the use of memory more carefully going forward with our object oriented programming designs.

More about calling and declaring functions

As seen, a *function* is a sequence of statements that can be *called*, accepting a sequence of *parameters*, and returning a result. Particularly because of the importance of types — and the fact that they are statically checked — functions must be declared before they are called; the declaration specifies not only the existence of the function, but also the types of parameters and the type of its return value.

Given this idea, you could write the definition of a function **square()** that takes an integer value and squares it.

```
int square(int num)
{
    return num * num;
}
```

Since function definitions include a signature, they also act as declarations, which would make the following program legal:

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

int square(int num)
{
    return num * num;
}

int main()
{
    cout << square(3) << endl;
    return 0;
}hu
```

The call to **square()** in **main()** is legal, because one *argument* that is an integer has been passed. The arguments listed when a function is called, in general, are matched up with the function's parameters in the order listed, with the type of each argument required to be compatible with the corresponding parameters.

Note, too, that order is important here: Since **square()** is defined before **main()**, it was legal to call **square()** from within **main().** But the reverse order would be problematic:

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

int main()
{
    cout << square(3) << endl;
    return 0;
}

int square(int num)
{
    return num * num;
}
```

The call to **square()** in **main()** would be deemed illegal by the compiler, because it precedes any declaration of the function.

However, remember that functions can be declared without being defined, so that allows a third possibility that would be legal:

```
include <iostream>
Using namespace std;

int square(int num);

int main()
{
    cout << square(3) << endl;
    return 0;
}

int square(int num)
{
    return num * num;
}
```

More about function return values, and the importance of heeding warnings

Most C++ functions return a single value of some type. While the idea that only one value can be returned sounds restrictive, it is important to realize that there are plenty of data types more complex than **int**, and that you will be able to define our own data types. A you can see, this restriction is quite workable in practice.

When you write a function that is said to return a value, you will want to be absolutely sure that *every path* through the function's statements reaches a **return** statement. While it is technically legal in C++ to write functions that do not explicitly return a value when they should, a well-configured compiler will at least generate what's called a *warning*, which will alert you to the very likely possibility that you have a bug. Warnings, generally, are messages emanating from your compiler that indicate the presence of a potential problem in code that is otherwise technically legal. For the most part, when you get a warning, it is indicating an issue you should be paying attention to.

For example, consider this function:

```
int absoluteValue(int num)
{
   if (num < 0)
   {
        return -n;
   }
}
```

At first glance, this function appears to do the important thing, which is to negate a negative input so that it becomes positive. However, consider what the function does when given a non-negative input. The conditional expression in the **if** statement will evaluate to **false**, but since there is no **else** branch, the function will not reach a **return** statement. So, what does the function return in this case?

Interestingly, the above function will actually compile successfully — on a typical compiler, you will receive a warning about not every path through the function leading to a **return** statement, but since this is not technically illegal in C++, you will not necessarily get an error. But its behavior will be undefined if you call it with anything except a negative input. For demonstration, you can write this short **main()** function to exercise this scenario:

```
#include <iostram>
using namespace std;
int main()
{
    cout << absoluteValue(-3) << endl;
    cout << absoluteValue(3) << endl;
    return 0;
}
```

Your programs in this course will be expected to compile both without errors *and without warnings*. no program can be compiled successfully and run if it causes any warnings to be emitted.

The fix in this example is straightforward: Make sure that every path through the function leads to a **return** statement that returns a value of type **int**.

```
int absoluteValue(int num)
{
```

```
    if (num < 0)
    {
         return -n;
    }
    else
    {
          return n;
    }
}
```

Functions that do not return a value

Note, too, that it is possible to write functions that do not return a value at all. You can do this by writing a function whose return type is listed as **void**. Such functions are generally useful for providing some side effect (such as writing output):

```
void printSquare(int num)
{
    cout << num * num << endl;
}
```

In these functions, you can write a **return** statement with no value, but it is not required; reaching the end of the function is sufficient to end its execution cleanly. A situation where you may need to write return statement sin void-returning functions is when an early bail out is called for.