PIC 10A: Introduction To Programming in C++

Michael Andrews
Program In Computing
UCLA Mathematics Department

December 4, 2024

Contents

1 About this document

I would like for future PIC 10A students to be given access to a free textbook upon enrolling in the class. By the end of this Summer Session A (8/16/2024), I hope that this document will have evolved into a rough, first draft of that textbook. It would be very unrealistic for me to attempt to write a complete textbook in 8 weeks and there will not be enough time for me to beautify all of my drawings from the supporting videos during this first attempt. Instead, during this quarter, I will aim for a concise resource that can continue to grow in future quarters. Conversations with you, the students of PIC 10A Summer Session A, will be particularly useful for informing me which sections need more detail and explanation, and which sections already successfully get the job done.

Integrated Development Environments (IDEs) often color your code to make it easier to read. In this document, I will color the code in a much simpler way. I hope that it will improve legibility, but I will provide a black and white version in case any of the color choices are inconvenient for some students.

2 Compilers and Integrated Development Environments

Coding languages have similarities and differences with spoken languages.

In a spoken language, grammatical mistakes may or may not infuence meaning. It is clear that "yesterday, I play football" means "yesterday, I played football." (The bigger issue is what is meant by "football". I am going to go with the sport where participants' feet regularly kick a ball!) On the other hand, "I like cooking my family and my pets," has a substantially different vibe to "I like cooking, my family, and my pets." In coding languages, grammatical errors almost always result in problems: the code might not run at all or it may perform its job incorrectly. Careful attention should be paid to the syntax of a coding language from the very beginning and questions about syntax are always good questions!

The English we speak developed over the last 1500 years or so and some words mean almost the opposite of what they used to: "terrific" used to mean terrifying; "nice" used to mean ignorant; "daft" used to mean accommodating; "silly" used to mean lucky. Moreover, when reading Shakespeare you will need to substitute "thou" and "thee" with "you", "thy" with "your", and "thine" with "yours". C++ strives to be a backward compatible language. This means that C++ code written in 1998 will still accomplish the same tasks successfully on a modern computer *providing that the code was well written.* Although newer syntax may have been introduced to the language, it does not affect the correctness of older syntax.

Spoken language is able to evolve so drastically because there are not official language police. Until a recent iOS update, iPhones would frequently change "its" to "it's" incorrectly (for example, in the sentence, "my cat checks its bowl even when it's not hungry") and no-one was taking Apple to court over this. On the other hand, part of a *compiler*'s job is to correctly enforce the rules of a coding language. Is everything definitely okay once your C++ code makes one compiler happy and your code runs correctly? Unfortunately not, and this issue requires some explanation...

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout << 1/0 << endl;
6      cout << 1/0 << endl;
7
8      int i = 1;
9      int j = 0;
10
11     cout << i/j << endl;
12
13     return 0;
14 }
```

⚠ Division by zero is undefined
⚠ Division by zero is undefined

Line: 15 Col: 1

```
1876948224
1
0
Program ended with exit code: 0
```

Many useful computer programs will need to perform millions of divisions (e.g. `8 / 4 == 2`). For this reason, it is necessary for divisions to be calculated as fast as possible. When you learned division, you were hopefully told that division by zero is undefined. Here is the important part: different computers use different hardware components and so it may be fastest for one computer to calculate `1 / 0 == 2`; it may be fastest for another computer to calculate `1 / 0 == 3`; and it may be fastest for another computer to give a different answer each time it calculates `1 / 0`. Because speed matters, it is a sensible choice to label division by zero as *undefined behavior*, and to allow different compilers to make different decisions regarding how divisions by zero are calculated. However, this has an unfortunate consequence. If you run your code using a computer and compiler that always says `1 / 0 == 0` *and you rely on this behavior*, then your code will behave completely incorrectly on a computer and compiler that has a different convention regarding division by zero. Although we have not learned any C++ yet, the screenshot on the previous page is provided to show XCode 15.4 (with default compiler settings) performing `1` divided by `0` three times and obtaining three different answers: `1876948224`, `1`, and `0`!

So what does a compiler do? A C++ compiler is given text files containing C++ code. Provided that the code passes the compiler's threshold, the compiler performs two processes called *compilation* and *linking* to produce an *executable*, a new type of file which will perform the instructions of the code when opened. All compilers should comply with a document called the C++ standard. The standard specifies that some code (e.g. division by zero) leads to undefined behavior, and this is why different compilers may have different thresholds, some being stricter than others regarding certain issues (Visual Studio 2022 does not compile the division-by-zero code above). However, all compilers will identify "incorrect grammar". None of them will forgive a missing semicolon, and they will not proceed beyond such an error.

In the very early days of coding, everyone used a text editor to write their code and the command line to compile, link, and execute their code. An Integrated Development Environment (IDE) is a piece of software to make this process, for lack of a better word, more integrated. Compilation, linking, and opening an executable is replaced by pressing a play button: much quicker! This is good because many new coding students are terrified of the command line and our IDEs will allow us to stay away from the command line for the duration of this class. Mac's most popular IDE is XCode and is downloadable from the App Store. You can see what it looks like above. Window's most popular IDE is Microsoft Visual Studio. Microsoft Visual Studio 2022 Community is available for download here. Visual Studio Code (a very similar name) is a different IDE that is also popular. Although it is a good IDE for Python, Java, and other languages, it has a higher startup cost for C++, and I would suggest choosing one of the previous two. Online GDB can also be a valuable resource.

Although our use of IDEs will mean that we can think much less about the complicated process (compilation, linking, opening an executable) going on behind the scenes, it is important to note that an IDE is different from a compiler. IDEs often support many different compilers, and changing compiler settings can impact how your code runs if your code includes undefined behavior. They can also influence how your code runs in good ways like optimizing your code to run faster!

On the following page we summarize the most important parts of this section.

2.1 Summary

- "Grammatical errors" are a big deal when coding.
- One part of a compiler's job is to check for grammatical errors.
  They will not forgive errors like missing semicolons.
- Compilers should comply with the C++ standard.
- Because of speed considerations and the existence of many different computers...
  - There are many different compilers.
  - The C++ standard specifies that some coding scenarios produce "undefined behavior".
  - Different compilers are allowed to handle undefined behavior in different ways.
  - This necessitates avoiding undefined behavior when writing C++ code.
- IDEs (like XCode and Visual Studio) allow us to compile, link, and run code with the press of a play button. This is the world in which we will live during this class.

3 Getting started: the shortest pieces of C++ code

Attempting to write the shortest piece of code that runs is a good place to start when learning any coding language, even if running is the only thing that the code manages to accomplish!

3.1 Doing "nothing"

Here is the shortest (ignoring spaces and line breaks) C++ code that will run.

```
1  int main() {
2      return 0;
3  }
```

This code defines a function called `main` that returns the `int`, `0`. This is a little unfortunate because we will only begin to properly discuss writing functions in section 9. `int`s, on the other hand, will be discussed much sooner and you can guess, to some extent, what they are. For now, I hope that the following explanation will suffice.

- A function will allow us to store a set of instructions.
- Often functions *return* a value, just like how $f(x) = 2x$ produces 8 when you feed it 4.
- When a C++ program executes, the set of instructions written in a function called `main` are executed. For this reason, there should be exactly one function called `main` per C++ program.
- Historically, the return value of the `main` function indicated the finishing status of the program and `0` meant "successfully finished".

In our case, we have written a program that communicates "program succeeded at doing nothing". Notice the numbers off to the left: 1, 2, 3. They are not a part of the code; they are line numbers. Notice the semicolon at the end of line 2. It is essential to the successful compilation of the code. Notice the four spaces at the start of line 2. These are not important for compilation or execution purposes, but they make the code easier to read and good use of *whitespace* is an important part of writing code.

3.2 Hello, World!

A "Hello, World!" program is often the first to be written by a new student learning a programming language. Such a program can also be used to check the correct installation of a new compiler and/or IDE.

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!\n";
5      return 0;
6  }
```

This program introduces a surprising number of new symbols, so let's discuss this example in detail.

We want the program to print "Hello, World!", but the words "Hello" and "World" are not part of the C++ language. They are simply sequences of characters that we want to end up in the console. In C++, we use double quotes to allow ourselves to type a *string literal*. The double quotes tell C++ compilers to interpret what's inside them as a sequence of characters and nothing more. Notice that in this document, when string literals are printed, spaces are replaced by a special character so that they are easily counted when there is more than one space: `"One␣Two␣␣Three␣␣␣"`.

In the string literal `"Hello,␣World!\n"`, the backslash and the `n` are not two characters. They are interpreted together as the *newline character*. Backslash is called an *escape character* and `\n` is called an *escape sequence*. Since we are on this topic, I will give two other examples. What if we want our string literal to contain some double quotes? How do we avoid the double quotes accidentally ending the string literal? `\"` is another escape sequence and so we can use the following string literal: `"Michael␣said,␣\"Backslash␣is␣an␣escape␣character␣in␣C++.\""`. What if we want a string literal to include a regular, non-escaping backslash? `\\` is an escape sequence and we can use the string literal, `"Michael␣said,␣\"\\␣is␣an␣escape␣character;␣\\\\␣is␣an␣escape␣sequence.\""`. To make you feel better about the overwhelming number of backslashes and to encourage you to run the code that I share with you here, these string literals are included as part of `snippet10-hello_world`.

Finally, we have to discuss `std::cout <<` and `#include <iostream>`.

`std::cout` is read as "stood-see-out", `std` stands for "standard", and `cout` stands for "character output". Think of `std::cout` like a machine that produces characters and places them in the console to be viewed by a user of the program. In order to produce characters, `std::cout` gobbles up various C++ entities which flow to it according to the arrows `<<`. `std::cout << "Hello,␣World!\n";` allows the string literal `"Hello,␣World!\n"` to be gobbled up by `std::cout` so that the characters it contains can be displayed in the console.

`#include` provides a way of including a standard-defined or user-defined file, and the files that are included in this way are called *header files*. The word "standard" refers to the C++ standard, so "standard-defined" means that the C++ standard specifies the definition, that is, what functionality should be provided by the header file. `<iostream>` is a standard library header which is part of the input/output library and it lets us use `std::cout` among many other things.

All of this sounds quite complicated. What is going on?! I'll answer this in a pragmatic way...

- We want to use `std::cout`. Whatever this is, the way it works must be "described" somewhere and we need to make sure that our program can make use of this "description".
- To do this we first type "`#include`".
- Then we google "c++ std::cout". Even though cpp reference is by far the best search result, this resource is too advanced for us at present, so you should follow the link to cplusplus.com.
- In the upper right of the page, you can see it says `<iostream>`. We can type or copy and paste this after `#include`.

We will learn how to write our own header files and include them with `#include` in section 13.

3.3 Hello, World! rewritten

The following code also successfully prints "Hello, World!"

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

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

There are some changes from the previous version which I'll now explain.

Typing `std::cout` all the time can seem a little ridiculous and begs the question: "why do we even have to type `std::`?" Many coding languages use the concept of a *namespace*. This is exactly what it sounds like - a space for names - but why would one want a space for names? For the same reason we have surnames and middle names and SSNs: to avoid name ambiguities. Suppose that you ask someone, "do you like Michael?" They will likely say, "who is Michael?" More relevant to this topic, they might also say, "which Michael do you mean?" There are too many Michaels: Michael Jordan, Michael B. Jordan, Michael Jackson, Michael Caine, Michael Douglas, Michael Keaton, Michael Schumacher, Michael Phelps, and the guy typing this! Similarly, in code there are words that we may want to use in different ways. The simplest example I can think of is "vector". You're not supposed to know very much C++ yet. In particular, it is unlikey that you know what a `std::vector` is, but you might have come across the word "vector" in mathematics. The only properties of mathematical vectors I want to mention is that they have a fixed dimension (or size) and that two vectors of the same dimension can be added. On the other hand, a `std::vector` can grow, that is, its size can change, and C++ does not provide us with a way to add `std::vector`s together. So, whatever a `std::vector` is, it is different than a vector in mathematics. If we were to try to define a data type for mathematical vectors in code, it would be useful for us to put it in a namespace called `math` and define the `math::vector` datatype. This would make it clear that `std::vector`s and `math::vector`s are different.

Having said all of that, if we are certain that we do not wish to define something with the same name as something else in the `std` namespace, then disambiguating is unnecessary. In this case, by writing `using namespace std`, we give compilers permission to find `std::cout` for us when we only type `cout`.

In the previous subsection, we thought of `cout` like a machine that produces characters in the console by gobbling up various C++ entities which flow to it according to the arrows `<<`. In fact, `cout` can gobble many entities in succession. For example, `cout << "Hello, World!" << endl` has `cout` gobble the string literal `"Hello, World!"` first; after this initial gobbling, `cout << endl` is left over and this has `cout` gobble an entity called `std::endl`. Gobbling `std::endl` is similar to gobbling `"\n"`, but it does one extra thing. I do not want to go into too much detail on this matter, and I am fine with you reaching the end of PIC 10A knowing that `std::endl` is different than `"\n"` even if you are not sure exactly why it is different, but here are some useful facts.

- If you are using print messages for progress updates and debugging purposes and your code is crashing for some reason, `endl` will make it more likely that you see the desired messages.

- Using `endl` will be equal to or slower than using `"\n"`, but if you only have a few `endl`s, then any slowdown will not be significant.

Because it is unlikely that you will use `endl` millions of times, these facts imply that using `endl` is fine for you at this stage of your coding career.

3.4 `snippet10-hello_world` and comments

When you download `snippet10-hello_world`, you will find it looks like this.

```
#include <iostream>
/*
 '#include' is used for including
 standard-defined or user-defined files.

 '<iostream>' is a standard library header file.
 It is part of the input/output library.
 Including it allows us to use std::cout.
*/

using namespace std;
/*
 'std' is a namespace.
 By writing "using namespace std",
 we give compilers permission to find std::cout
 even when we only type "cout".
*/


/*
 Each C++ program should have
 exactly one function called 'main'.

 For the first couple of weeks,
 almost everything that we type
 will be within the {}s of 'main'.

 When a C++ program runs,
 the set of instructions that are
 written in 'main' are executed.
*/
int main() {
    cout << "Hello, World!\n";
    cout << "Hello, World!" << endl;
    /*
     'cout' is like a machine that produces characters and
     places them in the console to be viewed by a user of the program.
     In order to produce characters, it gobbles up various C++ entities
     which flow to it according to the arrows '<<'.

     "Hello" and "World" are not part of the C++ language.
     They are simply sequences of characters that we want to end up in the console.
     In C++, we use double quotes to allow ourselves to type a 'string literal'.
     The double quotes tell C++ compilers to interpret what is inside them
     as a sequence of characters and nothing more.

     In the string literal "Hello, World!\n"
     the backslash and the n are interpreted together
     as the newline character. Backslash is called an
     'escape character' and \n is called an 'escape sequence'.
     Other examples of escape sequences are \" and \\.

     'endl' is similar to '\n' (but not exactly the same).
    */

    cout << "Michael said, \"Backslash is an escape character in C++.\""              << endl;
    cout << "Michael said, \"\\ is an escape character; \\\\ is an escape sequence.\"" << endl;
    // The code above demonstrates using \" and \\.


    return 0;
    /*
     Historically, the return value of the 'main' function was
     used to indicate the finishing status of the program and
     0 meant "successfully finished".
    */
}
```

By typing `//`, we can write a single-line *comment*. A multi-line comment can be created as follows...

```
/*
 A multi-line comment can be typed here.
*/
```

Compilers ignore comments and so the code in `snippet10-hello_world` executes identically to the following code.

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

int main() {
    cout << "Hello, World!\n";
    cout << "Hello, World!" << endl;

    cout << "Michael said, \"Backslash is an escape character in C++.\""              << endl;
    cout << "Michael said, \"\\ is an escape character; \\\\ is an escape sequence.\"" << endl;

    return 0;
}
```

Comments let us write helpful messages to other humans. Often the code featured in this document will contain fewer comments than the code shared on my website because this document contains fuller explanations of the code.

At least in this introductory section, you can see that the comments in `snippet10-hello_world` summarize many pages of discussion quite briefly.

3.5 Review questions

1. What restrictions are there on the number of `main` functions in a C++ program?
2. Why is `return 0` indented in the code that I have shared?
3. What is the output of the following code.

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

int main() {
    cout << "\\\n\"\n\\\\\"\n";
    return 0;
}
```

4. Can you write a program that prints `Hi!` without typing `using namespace std`?
5. Is using `\n` or `endl` faster? Does this influence your decision to use one or the other?

4 Errors

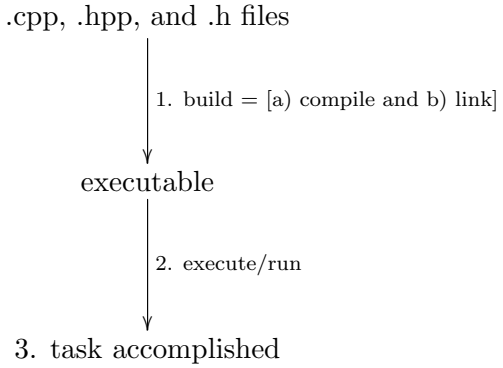
4.1 Metacognition

So far, we have only written variants on the "Hello, World!" program and it may feel strange to immediately turn our attention to what can go wrong when writing C++ code, but encountering errors is a very common aspect of coding in all languages. Having said that, I spend a small fraction of my coding-time dealing with unintentional errors. Spending too much time finding and fixing errors suggests an unproductive approach to coding. Here are some comments about errors, error messages, and the conceptual aspects of coding.

- **If your code has an error, it needs fixing immediately.** Once your code has two errors (or n errors), it is much more than twice (or n times) as difficult to fix them. One error at a time, please!!
- **Error messages can be daunting, but that is not a good reason to ignore them.** The more that you read error messages, the less daunting they'll become. Some error messages will even become familiar and you will be grateful when you encounter them because you will already know that you will be able to fix them quicker than less familiar errors.
- **Error messages provide an amazing learning opportunity.** I believe that at least 20% (probably more) of my time spent learning coding languages has consisted of making errors on purpose and trying to get to the bottom of how I'd violated the rules of the language. I'm saddened by the student who is happy their error has gone away when they have not taken the time to understand why they had an error in the first place. I know they will waste their time in the future on the same issue, and they will probably fail to answer the exam question that they just inspired. **Celebrate your errors by learning from them!**
- Your computer does not know your intentions and the suggestions your IDE provides you with may be shortsighted and conflict with your end goal. Furthermore, even simple typographical errors will often create problems for conceptual reasons. Error messages will highlight these conceptual reasons, and may say nothing about the possibility of a typographical error. Therefore, **your conceptual understanding of the language you are using to code strongly influences the readability of the error messages that you encounter and your speed at fixing the errors you come up against.** Many students argue that the conceptual aspects of a language feel unnecessary for their applications. Those students' applications might be really awesome, but they probably spend twice as long debugging them as they need to and they are likely to write code that is twice as long as it needs to be because they have not internalized powerful code-organizing concepts. **Learning the conceptual aspects of a coding language is a huge time-saver. Understanding the conceptual aspects is also important for scoring well on your coding exams.**

4.2 Build, execute, accomplish something

Suppose that you wish to write a C++ program to accomplish a specific task. You will type your C++ code into your IDE and then you will click on the play button. Clicking on the play button will trigger the start of a process. Ideally, the process will culminate with the specified task being accomplished successfully, but to understand what can go wrong, we need to highlight the different steps in the process.



1. The IDE will use a C++ compiler to attempt to compile and link your code.
   We will refer to this whole process as *building*.

2. If step 1 is successful, an executable will be produced.

   The IDE will save the executable on your computer somewhere, but it is not normally necessary for you to know where it is saved because the IDE will open it for you.

   At this point, your code will start to *execute* which is a fancy way to say "run".

3. If step 2 is successful, that is, your code finishes running, then a task has been accomplished. Hopefully, the correct task was accomplished!

Each step in this process can fail and this gives rise to some of the different errors we will consider.

4.3 Build errors

Suppose that step 1 of the process outlined in the previous subsection goes wrong. This means that we encounter a *build error*. Since building consists of compiling and linking, we actually encounter either a compile-time error or a linking error.

Linking errors are unpleasant and they normally only show up when using an IDE with multiple files (.cpp, .hpp, and/or .h files). We will not use multiple files for some time, so linking errors are much less relevant right now. However, there is a very easy way to create one: try hitting play on a completely empty text file!

Since we should not encounter many linking errors, most of our build errors will be compile-time errors. Even though the terminology "compile-time error" is more precise than "build error", we will continue to use the latter terminology.

Over the page, we give some examples.

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

int main() {
    cot << "Hello, World!\n";
    return 0;
}
```

There is a build error because `cot` is *undeclared* which means "not previously mentioned or known about". XCode is nice and provides the option to click "Fix" to change `cot` to `cout`. In general, you should be careful about clicking on the "Fix" button. There is at least one situation that previous students of mine have encountered where XCode's "fix" temporarily messes up their installation. While teaching PIC 10B, I accidentally demonstrated this midway through one of my lectures!

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

int main() {
    cout < "Hello, World!\n";
    return 0;
}
```

There is a build error because `cout` and `"Hello, World!\n"` cannot be compared using `<`. In this case, XCode does not suggest a fix.

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

int main() {
    cout << "Hello, World!\n"
    return 0;
}
```

This is a build error due to the missing semicolon.
This might be a common mistake for you before you build up some semicolon muscle memory.

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

int main() {
    return "Hello, World!\n";
}
```

There is a build error because a string literal is not an `int`.

We will see many more build errors as we learn more C++.

4.4 Runtime errors

Suppose that we do not encounter a build error, but the program stops midway through its execution due to an error. Such an error is called a *runtime error*.

The simplest example of a runtime error requires a little more than we have learned.

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

int main() {
    string s("0123");
    cout << s.at(8);

    return 0;
}
```

This example will make complete sense after we have discussed strings (different from string literals) in section 6. The issue is simple to describe though. There something called `s`. It has `4` characters, but we ask for the `8`-th character using a function called `at` and `at` is designed to hault a program when someone does something silly like this.

4.5 Undefined behavior

Remember that the C++ standard specifies that some coding scenarios produce "undefined behavior" and different compilers are allowed to handle undefined behavior in different ways.

- Undefined behavior could lead to a build error.
- Undefined behavior could lead to a runtime error.
- Most dangerously, code with undefined behavior could execute in a way that suggests there is no problem, when in reality, there is a nasty bug.
- More advanced... When compilers optimize code, they often assume that there is no undefined behavior. Code with undefined behavior may execute correctly until optimizations are turned on!

Earlier on in this document, I showed a screenshot of XCode 15.4 performing `1` divided by `0`. The code built and ran without encountering a runtime error even though it did produce strange results. A few years ago, when using an older version of XCode and an older Mac, the same code produced a runtime error. Undefined behavior is one of the most difficult aspects of C++. There is only one way to correctly "handle" it and that is to avoid it! Whenever I draw your attention to a scenario that produces undefined behavior, it is so that you can avoid that scenario like the plague.

One of the simplest and most common examples of undefined behavior is similar to the simplest example of a runtime error. This example will make complete sense after we have discussed strings

in section 6. `s` has `4` characters, but we ask for the `8`-th character using `operator[]` which is **not** required to hault a program (but still could) when someone does something silly like this.

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

int main() {
    string s("0123");
    cout << s[8];

    return 0;
}
```

4.6 Errors that are none of the above

It is possible that you have written C++ that compiles on all compilers, always finishes executing, has no undefined behavior, but does not accomplish the task you hoped. If it was our goal to greet the world, then the following code fails.

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

int main() {
    cout << "Hell␣World!" << endl;
    return 0;
}
```

It sounds more like lyrics from a heavy metal track. In fact, upon googling, I discovered that "Luke O'Neil's 'Welcome To Hell World' is a vital and despairing collection of essays on modern American life" and "Hellworld!" is also a story-driven 2D platforming game in which you possess your enemies and use their abilities to solve puzzles and make your way through the land of Hellworld.

It is also possible to write code that will never finish executing until it is forced to stop by the user. We will discuss this scenario when we learn about `while` loops in section 8, subsection 8.3.

4.7 Are some errors better than others?

While coding, build errors are the best because you often receive a reasonably informative message telling you how to fix your error. Runtime errors that arise while testing are the next best because these errors normally point you in the correct direction. Undefined behavior is definitely the worst and a special effort should be dedicated to avoiding it.

On the other hand, when turning in your homework assignments, build errors are the worst because they make it impossible to test your code, and therefore there is only one score that makes sense: `0`. Unfortunately, code that does not run is useless.

A summary of some of the preceding examples is provided in `snippet11-errors`.

4.8 Review questions

1. What are three things that you should do when you encounter an error?
2. When you click the play button in your IDE and no executable is created, what type of error have you encountered?
3. If you encounter a runtime error on your computer, does your code produce a runtime error on every other computer and compiler? Explain.
4. What type of error is your highest priority to avoid when turning in your homework solutions?

5 Variables and Fundamental Types

This is the section where we really get going with learning C++!

5.1 Types

Everything in C++ has a *type* and part of a compiler's job is to understand types and the ways in which they are allowed to interact with one another.

- As an example, we have already learned that `"Hello, World!"` is a *string literal*. More precisely, its type is `const char[14]`, but we will not discuss what this means until section 18.
- In order to emphasize that everything has a type, even when you do not want to think about it(!), we note that `std::cout` is a `std::ostream` and `std::endl` is a function (whose signature I'm choosing to omit). Streams are discussed more in PIC 10B, so it is very understandable if some of this feels intimidating right now (that is why I described `std::cout` in terms of "gobbling" instead), but it is good to be aware of where your understanding ends because then you know what to look up when you want to know more.
- Simpler examples of types are given by the following *fundamental types*: `int`, `double`, `char`, and `bool`. We will discuss these types extensively in this section and we will see that `4`, `3.2`, `'1'`, and `false` give examples of `int`, `double`, `char`, and `bool`, respectively.

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

int main() {
    cout << "  an int: " <<   4    << endl;
    cout << "a double: " <<  3.2   << endl;
    cout << "  a char: " <<  '1'   << endl;
    cout << "  a bool: " << false << endl;

    return 0;
}
```

5.2 Variables

For the "Hello, World!" program, we wrote several lines of code including the output; it would have been faster just to type the ouput! In order to write programs that perform useful tasks and really save us time, we need to introduce *variables*. The importance of variables is due to the following.

- A variable allows us to **store a value**.
- The value stored by a (non-`const`) variable **can be changed/updated**.
- Variables allow us to give **descriptive names** to useful values.

5.2.1 Declaring and defining variables

As mentioned above, everything in C++ has a type. In particular, every variable has a type. To give our first example of a variable, we will use the type `int`. This type allows us to store integers within a limited range. For now, all you need to know is that this data type allows us to store the numbers `-10000`, `-9999`, `-9998`, ..., `9998`, `9999`, `10000`. More details about the exact range of values that can be stored will be given shortly.

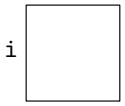
The following code builds and executes successfully to print `1`. I'll explain each line carefully.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i;
6
7      i = 1;
8
9      cout << i << endl;
10
11     return 0;
12 }
```

Definition 5.2.1.1. Line 5 is said to *declare and define* a variable called `i`. This tells the compiler...

- to reserve enough space in memory for an `int`;
- that the name `i` will be used to *access* and *modify* the value stored at that place in memory.

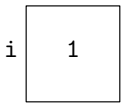
I find it useful to represent the consequences of line 5 graphically as a box labelled by `i`. The box is the visual representation of "some part of your computer's memory". For now, it is not important where such a value is stored in your computer.



Remark 5.2.1.2. In section 9, we will see that it is possible to declare a function without defining it. The same is true for a variable but doing this requires the `extern` keyword, something we will not learn about in this course. Therefore, when variables are being discussed, I do not mind if you mix up the words "declare" and "define". On the other hand, I will make sure that what I type is correct with respect to these subtleties. In case this issue ends up bothering you, the top answer to this Stack Overflow question is good.

Definition 5.2.1.3. Line 7 *assigns* the value of `1` to the variable `i`. This tells the compiler to store the value of `1` at the place in memory that was previously reserved by line 5.

In terms of the picture, you can think of this as putting `1` inside the box that was created by line 5.



Finally, line 9 asks to print the variable `i`. It is important to note that it is **not** asking to print the string literal `"i"`. This is an example of using a variable name to access its value, in this case the `int` `1`.

At this point there are already two build errors to learn about.

Build Error 5.2.1.4. *Using a variable before its declaration will result in a build error.*

Example 5.2.1.5. The following code gives a build error because `j` has not been declared.

```
1  int main() {
2      j = 1;
3
4      return 0;
5  }
```

Example 5.2.1.6. The following code gives a build error because `k` is used before its declaration.

```
1  int main() {
2      k = 1;
3      int k;
4
5      return 0;
6  }
```

Build Error 5.2.1.7. *Redefining a variable gives a build error.*

Example 5.2.1.8. The following code gives a build error because `i` is redefined on line 3.

```
1  int main() {
2      int i;
3      int i;
4
5      return 0;
6  }
```

`snippet12-declaring_defining` summarizes this subsection.

5.2.2 The assignment operator

Definition 5.2.2.1. `=` is called the *assignment operator*.

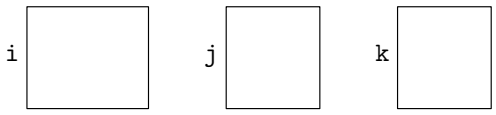
The assignment operator is used to update the value of a variable written on the left side of it by evaluating the expression written on the right side of it.

Remark 5.2.2.2. It is important that this operator does not test for equality. There is an "equal to" operator for that.

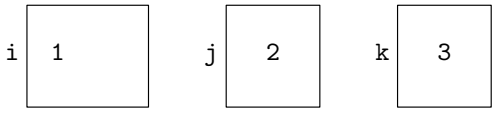
Example 5.2.2.3. Consider the following code.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i, j, k;
6
7      i = 1; j = 2; k = 3;
8      cout << i << "␣" << j << "␣" << k << endl;
9
10     i = j;
11     cout << i << "␣" << j << "␣" << k << endl;
12
13     i = j + k;
14     cout << i << "␣" << j << "␣" << k << endl;
15
16     return 0;
17 }
```

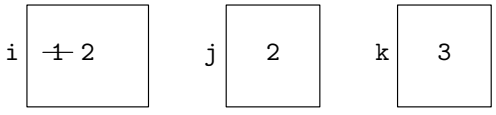
Line 5 declares (and defines) three variables simultaneously.



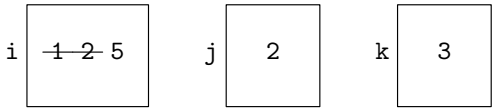
Line 7 makes three assignments.



Line 10 says `i = j`.
The expression on the right is `j` and `j`'s value is `2`.
The variable on the left is `i`, so `i`'s value is updated to be `2`, overwriting its previous value of `1`.



Line 13 says `i = j + k`.
The expression on the right is `j + k`. `j`'s value is `2`. `k`'s value is `3`. `j + k` is evaluated to give `5`.
The variable on the left is `i`, so `i`'s value is updated to be `5`, overwriting its previous value of `2`.



The output of the code shows the status of the variables after each line that involves an assignment...

```
1 2 3
2 2 3
5 2 3
```

Because the assignment operator is used to update the left side using the right side, it is asymmetric (unlike equality). Therefore, descriptions involving it will also be asymmetric, and language needs to be used carefully in order to express one's thoughts correctly. Consider the assignment used on line 13 of the previous example: `i = j + k`.

- `i` is referred to as the *assigned-to* variable.
- `j + k` is referred to as the *assigned-from* expression.
- If I am narrating as I write the code or I am instructing you to type it, I will say, "`i` assign `j` plus `k`."
- When describing what the line of code accomplishes, I will say
  - either "the value of `j + k` is assigned to `i`";
  - or "`i` is assigned the value of `j + k`".

  I like the first description because the preposition "to" explicitly indicates the direction that information is flowing: `j` and `k`'s values are being used to influence `i`'s value. It is also listed in wiktionary's definition of "assign". I like the second description a little less, but it has the benefit that `i`, `j`, and `k` are said in the same order in which they are typed.

  The following is an incorrect of use language: "`i` is assigned to the value of `j + k`".

  I have encountered many students saying something like this. To see why this is incorrect, replace `i` by "Michael", "assign" by "throw", `j + k` by "tennis ball" and change to the past tense (so that everything reads a little more fluidly). The correct sentences become...
  - "the tennis ball was thrown to Michael";
  - "Michael was thrown the tennis ball".

  The incorrect sentence becomes "Michael was thrown to the tennis ball".

  `i`'s value is being influenced; Michael's possession of the tennis ball is being influenced.

5.2.3 Initializing variables

Definition 5.2.3.1. The first assignment of a value to a variable is said to *initialize* the variable.

Undefined Behavior 5.2.3.2. *Using a variable before it is initialized gives undefined behavior.*

Example 5.2.3.3. The following code encounters undefined behavior because we print `ub` on line 7 before its initialization on line 9.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int ub;
6
7      cout << ub << endl;
8
9      ub = 1;
10
11     return 0;
12 }
```

This example emphasizes the danger of undefined behavior.

- At the time of writing, Online GDB prints `0` upon executing this code.
- In the past, XCode **used to** print `0` upon executing this code.

Therefore, in the past, OnlineGDB and XCode might have led you to think that such code reliably prints `0`. However...

- At the time of writing, XCode does **not** reliably print `0`.
- At the time of writing, Visual Studio does **not** even build the code.

To avoid undefined behavior 5.2.3.2, it can be useful to initialize a variable at the same time that it is declared. The following code shows this in action. Line 5 declares, defines, and initializes the variable `p`. Line 6 declares, defines, and initializes the variables `q` and `r`. The output is `1 2 3`.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int p = 1;
6      int q = 2, r = 3;
7
8      cout << p << "␣" << q << "␣" << r << endl;
9      return 0;
10 }
```

`snippet13-assignment_init` summarizes the last two subsections.

5.2.4 Naming rules and conventions

Each variable needs a name that identifies it and distinguishes it from others (in the same scope). Valid *identifiers* begin with a letter or an underscore. They continue with any number of additional characters that are letters, digits, or underscores, but they cannot be keywords like `int` or `double`. C++ is case-sensitive: `variable`, `Variable`, and `VARIABLE` are regarded as distinct identifiers.

There are few common conventions for identifiers that consist of more than one word:

- `ALL_CAPS_SEPARATED_BY_UNDERSCORES`
- `words_which_May_or_may_not_start_with_a_Capital_separated_by_underscores`
- `UpperCamelCase`
- `lowerCamelCase`

I think that everyone agrees that constants should use `ALL_CAPS_SEPARATED_BY_UNDERSCORES`. I have seen all of the other conventions used in various contexts in different languages. For non-constant variables in C++, I prefer to use underscores. The most common "other choice" is `lowerCamelCase`. It is more important to be consistent than to try and decide upon a best choice. Most experienced coders would become unhappy while reading the following code.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int one_variable = 0;
6      int anotherVariable = 1;
7      int ChangingConvention = 2;
8      return 0;
9  }
```

5.2.5 Review questions

1. (a) How do you declare and define an `int` called `my_int`?
   (b) How do you make `my_int` store `8`?
   (c) Draw a picture that represents what the previous two questions accomplish.

2. How many and what type of errors does the following code produce?

```
1  int main() {
2      a = 0;
3      int a;
4      int a;
5
6      return 0;
7  }
```

3. Suppose that `int`s called `x`, `y`, and `z` have been declared and consider the code: `x = y * z`.
   (a) How would you instruct someone to type `x = y * z`?
   (b) What word should you **not** say?
   The answer to this might surprise someone who just started to code one minute ago.
   (c) Write a sentence that:
   - describes what this line of code accomplishes;
   - uses the word "assign" correctly.

   Say it aloud a few times.

4. What happens when you print an `int` before initializing it?

5. How are you going to name your more-than-one-word variables?
   - `using_underscores`
   - `lowerCamelCase`
   - `a_bit_ofBoth`

5.3 int

5.3.1 +, -, *, +=, -=, *=

ints can be added, subtracted, and multiplied by using +, -, and *, respectively. Lines 7, 8, and 9 of the following code give examples of adding, subtracting, and multiplying two variables i and j. Line 14 shows that these calculations do not influence the values of i and j: i and j still have the values of 3 and 2, respectively, that they were initialized with on line 5. If we want to influence the values of i and j, we should use the assignment operator and/or the asymmetric operators +=, -=, and *=. For example, we know line 17 updates i's value to be 2.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i = 3, j = 2;
6
7      cout << "i␣+␣j␣is␣" << i + j << endl; // 5
8      cout << "i␣-␣j␣is␣" << i - j << endl; // 1
9      cout << "i␣*␣j␣is␣" << i * j << endl; // 6
10
11     cout << endl;
12
13     cout << "i" << "|" << "j" << endl;     // i|j
14     cout <<  i  << "|" <<  j  << endl;     // 3|2 (still)
15
16
17     i = j;
18
19     cout <<  i  << "|" <<  j  << endl;     // 2|2
20
21     i = i + 3;
22     j += 3;
23
24     cout <<  i  << "|" <<  j  << endl;     // 5|5
25
26     i = i - 3;
27     j -= 3;
28
29     cout <<  i  << "|" <<  j  << endl;     // 2|2
30
31     i = i * 3;
32     j *= 3;
33
34     cout <<  i  << "|" <<  j  << endl;     // 6|6
35
36
37     return 0;
38 }
```

Line 21 looks a bit weird because `i` appears on the left and right hand side and the mathematical equation $i = i + 3$ does not have a solution in the real numbers, but the assignment operator is **not** equality. The expression on the right of `i = i + 3` is `i + 3` which evaluates to `5`, and so the value of `5` is assigned to `i`. This effectively increases `i`'s value by `3`. Even though I understand it, I do not like the look of `i = i + 3`. The next line `j += 3` increases `j`'s value by `3`. It is essentially a rewriting of the code `j = j + 3`, but I prefer the syntax in this context. Lines 26 and 27 and lines 31 and 32 show you similar operators. `j -= 3` says to decrease `j`'s value by `3`, and `j *= 3` says to multiply `j`'s value by `3`.

5.3.2 `++`, `--`

By themselves, `++i` and `i++` both behave like `i += 1`, but they act a little differently if they show up on the right hand side of an assignment. You can think of `++i` as meaning "increment and then assign", whereas `i++` means "assign and then increment". More explicitly, `ppi = ++i` behaves like `i += 1; ppi = i` whereas `ipp = i++` behaves like `ipp = i; i += 1`. The following code demonstrates this.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i, ppi, ipp;
6
7      i = 0;  ++i;
8      cout <<   i;  // 1
9
10
11     i = 0;  i++;
12     cout << i  ;  // 1
13
14
15     i = 0;  ppi = ++i;
16
17     cout <<   i;  // 1
18     cout << ppi;  // 1, the value after incrementing i
19
20
21     i = 0;  ipp = i++;
22
23     cout << i  ;  // 1
24     cout << ipp;  // 0, the value before incrementing i
25
26
27     cout << endl;
28
29     return 0;
30 }
```

It is unlikely you will want to make heavy use of `++` together with assignments. However, because

many coders do like to use `++` in this way it was important to explain how `++i` and `i++` differ. If you are not using `++` with assignments, it makes little difference whether you use `++i` or `i++`. However, I have a preference for `++i` because it is likely to be the faster operation in other contexts.

`--i` and `i--` behave similarly to `++i` and `i++`, but with `i -= 1` replacing `i += 1`.

Undefined Behavior 5.3.2.1. *Using `++i` twice within one code block can give undefined behavior.*

Example 5.3.2.2. The following code outputs `12` on XCode 15.4 and OnlineGDB, but `22` on Visual Studio 2022. This is because the order of `cout`'s gobbling and the two `++i`s is unspecified by the C++ standard.

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

int main() {
    int i = 0;
    cout << ++i << ++i << endl;

    return 0;
}
```

5.3.3 `int`-division, `/`, `%`, `/=`, `%=`

Dividing one `int` by another `int` gives an `int`. This can be surprising at first, but I promise you that there are a number of contexts where this is a useful language decision. What does this division do? The division that you learned about before you knew about fractions.

Example 5.3.3.1. `28` divided by `10` is `2` with a remainder of `8` because $28 = 2 \cdot 10 + 8$.

Upon reading this example we recall a lost friend: the *remainder*. On the other hand, the C++ operation `/` never forgot its best friends: `%`, `/=`, and `%=`.

- `%` calculates the remainder of the corresponding calculation with `/`;
- `i /= j` and `i %= j` are like `i = i / j` and `i = i % j`, respectively.

Negative `int`s can create a little confusion. Because of them, it can be useful to consider another description which can serve as a definition...

Definition 5.3.3.2.

- For two `int`s, `i` and `j`, `i / j` is also an `int`. In this case, `i / j` is said to perform `int`-*division*.

  The integer calculated by `i / j` is obtained by dividing the integers stored by `i` and `j` using normal mathematics and then discarding the fractional part of the result.

- For two `int`s `i` and `j`, `i % j` calculates `i - ((i / j) * j)`.

  This is a formula for the remainder when performing `i / j`.

Example 5.3.3.3.

- When `i` stores `28` and `j` stores `10`, `i / j` is `2`.
  This is because $\frac{28}{10} = 2\frac{4}{5}$ and we discard the $\frac{4}{5}$ when performing `int`-division.
- When `i` stores `28` and `j` stores `10`, `i % j` is `8` because $28 - (2 \cdot 10) = 8$.
- When `i` stores `-28` and `j` stores `10`, `i / j` is `-2`.
  This is because $\frac{-28}{10} = -2\frac{4}{5}$ and we discard the $-\frac{4}{5}$ when performing `int`-division.
- When `i` stores `-28` and `j` stores `10`, `i % j` is `-8` because $-28 - (-2 \cdot 10) = -8$.
  This negative remainder has surely caught out some excellent coders!

Here is some code based on the previous example.

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

int main() {
    int i;
    int j;


    i = 28; j = 10;

    cout << i      << " divided by "      <<      j << " is ";
    cout << i / j << " with remainder " << i % j <<   endl;


    i = -28; j = 10;

    cout << i      << " divided by "      <<      j << " is ";
    cout << i / j << " with remainder " << i % j <<   endl;


    return 0;
}
```

Undefined Behavior 5.3.3.4. *`int`-division by `0` gives undefined behavior.*

Here is some code to demonstrate `/=` and `%=`. Notice that by using these operations with powers of `10` we can delete digits from an `int`.

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

int main() {
    int i;


    i = 2345;
    cout << "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣i␣is␣" << i << endl; // 2345

    i /= 10;
    cout << "after␣i␣/=␣10,␣i␣is␣" << i << endl; // 234

    i /= 10;
    cout << "after␣i␣/=␣10,␣i␣is␣" << i << endl; // 23

    i /= 10;
    cout << "after␣i␣/=␣10,␣i␣is␣" << i << endl; // 2

    i /= 10;
    cout << "after␣i␣/=␣10,␣i␣is␣" << i << endl; // 0


    cout << endl;


    i = 2345;
    cout << "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣i␣is␣" << i << endl; // 2345

    i %= 1000;
    cout << "after␣i␣%=␣1000,␣i␣is␣" << i << endl; // 345

    i %= 100;
    cout << "␣after␣i␣%=␣100,␣i␣is␣" << i << endl; // 45

    i %= 10;
    cout << "␣␣after␣i␣%=␣10,␣i␣is␣" << i << endl; // 5

    i %= 1;
    cout << "␣␣␣after␣i␣%=␣1,␣i␣is␣" << i << endl; // 0


    return 0;
}
```

5.3.4 What *is* an `int`?

You're not going to like the answer to this. What an `int` *is* depends on the computer and compiler. Sigh. However, *every* computer and compiler I have ever encountered (which includes greater than 1000 students' computers) has agreed on what an `int` is. So let's simply things by assuming this is the only scenario that one can ever encounter. C++20 (the version of C++ released in 2020) has also made sure that what I am saying is closer to the full truth than ever before!

"Under the hood" is a phrase that is often used to describe details that are obscured from many users' interaction with a technical thing.

Definition 5.3.4.1. Under the hood, an `int` consists of 32 `0`s and `1`s which are used to encode an integer between -2^{31} and $2^{31}-1$. If $b_0, b_1, b_2, \ldots, b_{29}, b_{30}, b_{31}$ are 32 values, each `0` or `1`, then they encode the integer:

$$b_0 \cdot 2^0 \ + \ b_1 \cdot 2^1 \ + \ b_2 \cdot 2^2 \ + \ b_3 \cdot 2^3 \ + \ \ldots \ + \ b_{28} \cdot 2^{28} \ + \ b_{29} \cdot 2^{29} \ + \ b_{30} \cdot 2^{30} \ - \ b_{31} \cdot 2^{31}$$

The negative sign before $b_{31} \cdot 2^{31}$ is not a typographical error and it is the only negative sign in the formula. Another way to explain this encoding is that an integer between 0 and $2^{31}-1$ is encoded using *binary*, and an integer i between -2^{31} and -1 is encoded by expressing $i + 2^{32}$ in binary. The numbers $b_0, b_1, b_2, \ldots, b_{29}, b_{30}, b_{31}$ are called *bits*.

Example 5.3.4.2. The `int` `0` is expressed using 32 `0`s and the `int` `-1` is expressed using 32 `1`s. This is because...

$$0 \ = \ 0 \cdot 2^0 \ + \ 0 \cdot 2^1 \ + \ 0 \cdot 2^2 \ + \ 0 \cdot 2^3 \ + \ \ldots \ + \ 0 \cdot 2^{28} \ + \ 0 \cdot 2^{29} \ + \ 0 \cdot 2^{30} \ - \ 0 \cdot 2^{31} \text{ and}$$
$$-1 \ = \ 1 \cdot 2^0 \ + \ 1 \cdot 2^1 \ + \ 1 \cdot 2^2 \ + \ 1 \cdot 2^3 \ + \ \ldots \ + \ 1 \cdot 2^{28} \ + \ 1 \cdot 2^{29} \ + \ 1 \cdot 2^{30} \ - \ 1 \cdot 2^{31}.$$

The `int` `88` is expressed by `00011010 00000000 00000000 00000000` because

$$88 \ = \ 0 \cdot 2^0 \ + \ 0 \cdot 2^1 \ + \ 0 \cdot 2^2 \ + \ 1 \cdot 2^3 \ + \ 1 \cdot 2^4 \ + \ 0 \cdot 2^5 \ + \ 1 \cdot 2^6 \ + \ 0 \cdot 2^7 \ + \ \text{lots of 0s}$$

The `int` `-2147483600` is expressed by `00001100 00000000 00000000 00000001` because

$$-2147483600 \ = \ 48 - 2147483648 \ = \ (2^4 + 2^5) - 2^{31}.$$

These details are mainly provided for completeness even though they can have fun applications. In fact, when computers were less powerful, details like these were used in the early development of computer games. We will study binary to some extent in this class, but it is not hugely important for you to remember how `int`s are stored under the hood. It *is* important for you to remember that `int`s have a limited range.

- There is a minimum `int`: `-2147483648`.
- There is a maximum `int`: `2147483647`.

5.3.5 Review questions

1. The following code builds and runs successfully on all compilers.

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

int main() {
    int a = 2;
    int b = 3;

    cout <<     a << "␣" << b     << endl;

    a += 4;
    b *= 5;

    cout <<     a << "␣" << b     << endl;
    cout << b / a << "␣" << b % a << endl;
    cout <<     a << "␣" << b     << endl;

    b %= a;
    a %= b;

    cout <<     a << "␣" << b     << endl;

    return 0;
}
```

   What is the output when the code executes?

5.4 const

When writing code, it is good to try and avoid using *magic numbers*. For example, if the number of hours in a day is relevant in some code that you are writing, it is best not to use the `int` `24` (the magic number in this example) in your code without explanation. Instead it is better to declare a `const int` called `HOURS_PER_DAY`. `const` helps protect yourself and the people who use your code against accidentally changing a number that is supposed to be a constant.

`int` is an example of a *fundamental type*...

Build Error 5.4.1. *If a variable's type is a `const` fundamental type, the variable must be initialized when it is defined.*

Build Error 5.4.2. *If a variable's type is a `const` fundamental type, assigning to the variable after initialization gives a build error.*

Example 5.4.3.

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

int main() {
    const int HOURS_PER_DAY = 24;

    /*
     Uncommenting the next line would produce a build error.
     Because HOURS_PER_DAY is a 'const int', assigning to
     the variable after initialization gives a build error.
    */
    // HOURS_PER_DAY = 0;

    const int MINUTES_PER_HOUR = 60;
    const int MINUTES_PER_DAY  = MINUTES_PER_HOUR * HOURS_PER_DAY;

    cout  << "MINUTES_PER_DAY␣is␣" << MINUTES_PER_DAY << endl;

    /*
     The next line would produce a build error.
     Because SECONDS_PER_DAY is a 'const int',
     it must be initialized when it is defined.
    */
    // const int SECONDS_PER_DAY;

    return 0;
}
```

As one learns more C++, eventually one realizes how amazing `const` really is. A build error due to violating a rule concerning `const` often reveals mistaken intentions and saves a real headache.

5.5 std::cin

We have seen how to print information to the console. We also need a way to enter information in the console. This will allow, for example, a simple conversation between us and the computer.

Just as there is std::cout, a machine that produces characters and places them in the console to be viewed by a user of the program, there is also std::cin, a machine that extracts characters from the console. While doing this, it also feeds variables with values along the arrows >>. This demonstrates some cleverness on operator>>'s part because it can do this for many of the different types that we will learn about.

Example 5.5.1. When the following code executes, the user will be able to respond to the question in the console. If they type a sequence of characters which is appropriately numerical, the code will execute as though a value was assigned to i. For example, if they type the characters 1234 and hit the ENTER key, the code will execute as though the int 1234 was assigned to i.

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

int main() {
    cout << "Hi! What's your favorite int?" << endl;

    int i;
    cin >> i;

    cout << "Oh wow! " << i << " is my favorite too." << endl;
    cout << "What are the chances of that?"           << endl;

    return 0;
}
```

Example 5.5.2. In this example, if the user types 111 222 and hits the ENTER key, the code will execute as though the int 111 was assigned to i1 and the int 222 was assigned to i2.

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

int main() {
    cout << "Hi! Give me two ints separated by a space." << endl;

    int i1, i2;
    cin >> i1 >> i2;

    cout << "Their sum is " << i1 + i2 << "." << endl;
    return 0;
}
```

We will discuss how cin >> works in much more detail in section 7. This will allow us to understand what happens when the user of the program is less good at responding to the prompts!

5.6 bool

bool, the Boolean data type, is a fundamental type named after George Boole. Booleans have one of two possible values: false and true. Awkwardly, by default, printing a bool with cout << will lead to seeing either 0 or 1 in the console, not false or true.

Example 5.6.1. The output of the following code is 01.

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

int main() {
    bool b;

    b = false; cout << b;
    b = true;  cout << b;

    cout << endl;
    return 0;
}
```

You may prefer to see false and true in the console. This can be accomplished by feeding std::cout with std::boolalpha before using std::cout to print a bool.

Example 5.6.2. The output of the following code is: false true.

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

int main() {
    bool b;
    cout << boolalpha;

    b = false; cout << b << "␣␣";
    b = true;  cout << b << "\n";

    return 0;
}
```

Booleans show up frequently as the result of comparisons and == is the most famous of all the comparison operators.

Definition 5.6.3. == is called the *equality operator* or the *"equals to" operator*.
For fundamental types, the equality operator is used to compare...

- the value obtained by evaluating the expression written on the left side of it with...
- the value obtained by evaluating the expression written on the right side of it.

Example 5.6.4. In the following code `2 == 2` evaluates to `true` and is assigned to `b` on line 8.
`2 + 2 == 5` evaluates to `false` and is assigned to `b` on line 11.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      bool b;
6      cout << boolalpha;
7
8      b = (2 == 2);
9      cout << "(2 == 2) is " << b << "." << endl;
10
11     b = (2 + 2 == 5);
12     cout << "(2 + 2 == 5) is " << b << "." << endl;
13     cout << "Sorry, Radiohead"       << "." << endl;
14
15     return 0;
16 }
```

The output of the code above is

```
1  (2 == 2) is true.
2  (2 + 2 == 5) is false.
3  Sorry, Radiohead.
```

Example 5.6.5. The following code asks a user for an `int`.
It uses the remainder upon division by `2` to correctly tell them whether their `int` is even or not.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout << "Hi! Give me an int." << endl;
6
7      int i;
8      cin >> i;
9
10     bool i_is_even = (i % 2 == 0);
11
12     cout << boolalpha;
13
14     cout << "It is " << i_is_even;
15     cout << " that " << i << " is even." << endl;
16
17     return 0;
18 }
```

5.7 double

5.7.1 Arithmetic operations and double-division

We have seen that the fundamental type int is good for storing integers within a range. But what about numbers with a fractional part. That is where the fundamental type double becomes useful. "Double" stands for double-precision floating-point format.

Example 5.7.1.1. We have the expected arithmetic operations: +, -, *, /. We have the associated asymmetric operators +=, -=, *=, /=, and we have pre-++, post-++, pre--, and post--. The following code demonstrates the most basic operations.

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

int main() {
    double d1 = 1.25;
    double d2 = 0.5;

    cout << d1 + d2 << endl;
    cout << d1 - d2 << endl;
    cout << d1 * d2 << endl;
    cout << d1 / d2 << endl;

    return 0;
}
```

The output is as follows.

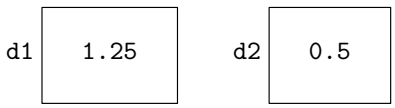
In particular, 1.25 / 0.5 == 2.5, so division accounts for the fractional parts of the numbers.

```
1.75
0.75
0.625
2.5
```

Definition 5.7.1.2. When d1 and d2 are doubles, d1 / d2 is said to perform double-*division*.

Undefined Behavior 5.7.1.3. *double-division by 0.0 gives undefined behavior.*

What is the picture for the code above? It is as if we had two ints. However, the fractional parts of the numbers indicate that we are talking about doubles. In reality, the boxes *are different*. For one thing, on most modern computers, ints will be stored using 4 bytes, whereas doubles will be stored using 8 bytes. The boxes represent chunks of memory of different sizes, and we cannot fit a double into a box that was meant for an int.



5.7.2 Casting between `int`s and `double`s

`int` is used to store integers. `double` is used to store numbers with a fractional part. But wait...

- An integer can be thought of a number whose fractional part is zero.
- Starting with a number with a fractional part and discarding it gives an integer.

For these two reasons, C++ allows *casting* from `int` to `double` and from `double` to `int`.

Definition 5.7.2.1.

- If `i` is an `int`, then `static_cast<double>(i)` is a `double` with zero fractional part whose integer part agrees with the original `int`.
- If `d` is a `double`, then `static_cast<int>(d)` is an `int`. As long as a `double` `d` is not too large, `static_cast<int>(d)` is the result of discarding the fractional part from `d`. This might be referred to as *truncation*.

Example 5.7.2.2. In this example, the `int` `2` is used to create the `double` `2.0`, and the `double` `4.8` has its fractional part discarded to give `4`.

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

int main() {
    int    i = 2;
    double i_with_zero_fractional_part = static_cast<double>(i);

    double        d = 4.8;
    int truncated_d = static_cast<int>(d);

    cout << i                            << "␣" <<           d << endl;
    cout << i_with_zero_fractional_part << "␣" << truncated_d << endl;

    return 0;
}
```

The output is ...

```
2 4.8
2 4
```

We see that nothing about the default way `2.0` is printed indicates that it is a `double`, but we know!

Remark 5.7.2.3. C is the language that came before C++ (you understand the name now). There was some syntax for casting in C. It is better not to use C-style casting when coding in C++, so I am not going to tell you the syntax.

5.7.3 Mixed division

Definition 5.7.3.1.

- Dividing two `int`s performs `int`-division to give an `int`.
- Dividing two `double`s performs `double`-division to give an `double`.
- If `i` is an `int` and `d` is a `double`,
  - `i / d` calculates `static_cast<double>(i) / d`
  - `d / i` calculates `d / static_cast<double>(i)`

  so both calculations give a `double`.

Example 5.7.3.2. In the following example, the first numerical output is the result of `int`-divison. If the intension was to calculate the average (mean) of the three values, then it fails when the user enters `0 0 1` giving `0` instead of `0.333333`.

The second numerical output is the result of `double`-divison.

The third numerical output comes from `sum / 3.0`. Dividing by `3.0`, which is a `double`, causes `sum` to be cast to a `double`, `static_cast<double>(sum)`, and so the output is the same as the previous output.

The final output comes from `static_cast<double>(sum) / 3`. Since `static_cast<double>(sum)` is a `double`, this causes `3` to be cast to the `double` `3.0`, and so the output is the same as the previous output again.

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

int main() {
    cout << "Type␣three␣integers␣separated␣by␣spaces:␣";

    int       i1,  i2,  i3;  cin >> i1 >> i2 >> i3;
    int sum = i1 + i2 + i3;

    cout <<                       sum  / 3   << endl;
    cout << static_cast<double>(sum) / 3.0 << endl;
    cout <<                       sum  / 3.0 << endl;
    cout << static_cast<double>(sum) / 3   << endl;

    return 0;
}
```

5.7.4 `<cmath>`

Now that we have `int`s, `double`s, and `cin >>`, we can write useful programs. We have seen that we can perform many mathematical operations. By using the `<cmath>` header, we can also use common mathematical functions.

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

int main() {
    cout <<   pow(2, 8) << endl; // 2 to the power of 8 is 256
    cout <<  sqrt(9)    << endl; // the square root of 9 is 3
    cout <<   cos(0)    << endl; // cosine of 0 is 1
    cout <<  acos(-1)   << endl; // arccosine of -1 is PI

    cout << round(17.8) << endl; // 18
    cout << round(18.2) << endl; // 18
    cout << round(-3.3) << endl; // -3
    cout << round(-3.5) << endl; // -4

    return 0;
}
```

I have just told you that including <cmath> will allow us to use all the functions std::pow, std::sqrt, std::cos, std::acos, and std::round, but if you didn't know this, you might have guessed a function called round exists.

- Google "c++ round". Even though cpp reference is by far the best search result, this resource is too advanced for us at present, so follow the link to cplusplus.com.
- The page confirms that round exists. Moreover, in the upper right of the page, you can see it says <cmath> and so we should #include this header. (It also says <ctgmath>, but one #include is enough, and since cpp reference says nothing about <ctgmath>, <cmath> is the better choice.)

5.7.5 What *is* a double?

Just as for ints, what a double *is* depends on the computer and compiler. However, *every* computer and compiler I have ever encountered in the last 5 years has agreed that doubles are stored using 8 bytes. Then, ignoring some particularly exotic doubles (*signed zero*, *subnormal numbers*, ∞ and NaNs) we can give a fairly concise description of them. The Wikipedia Page is an excellent account on double-precision floating-points if you want more details.

Definition 5.7.5.1. Under the hood, a double consists of 64 0s and 1s grouped together:

- 1 sign bit, s,
- 11 exponent bits, $e_0, e_1, \ldots, e_9, e_{10}$,
- 52 significand bits, $f_0, f_1, \ldots, f_{50}, f_{51}$.

Ignoring a few special cases, these values encode the number $(-1)^s \cdot 2^E \cdot (1+F)$ where:

- $E = -1023 + \left[e_0 \cdot 2^0 + e_1 \cdot 2^1 + e_2 \cdot 2^2 + \ldots + e_9 \cdot 2^9 + e_{10} \cdot 2^{10}\right]$;
- $F = \frac{1}{2^{52}} \cdot \left[f_0 \cdot 2^0 + f_1 \cdot 2^1 + f_2 \cdot 2^2 + \ldots + f_{50} \cdot 2^{50} + f_{51} \cdot 2^{51}\right]$.

What can we takeaway from this complicated description?

- $0 \leq F < 1$, so these numbers are expressed like scientific notation, but with 2^E replacing 10^E.
- After a shift, the *exponent* E is expressed in binary, and
  after a scaling, the same is true for the *significand* F.
- Not every number can be written this way.
  Therefore **not all numbers can be stored using the `double` fundamental type.**
  This is the only part that you need to know for examination purposes in PIC 10A.

5.7.6 Precision of `double`

Because not all numbers can be stored using the `double` fundamental type, many numbers have to be approximated before they are stored, and this has observable consequences.

Example 5.7.6.1. Consider the following code.

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

int main() {
    double d = 4.0 / 3.0 - 1.0 - 1.0 / 3.0;

    cout << boolalpha;

    cout << (d == 0.0)           << "␣";   // false
    cout << (d <  0.0)           << "␣";   // true

    cout << (d == -pow(2, -54)) << endl; // true

    return 0;
}
```

The output of `false true true` may be surprising to you, but if you know the mathematics in Math 31B, the output can be explained using the following formula.

$$\frac{4}{3} = 1 + \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^8} + \frac{1}{2^{10}} + \cdots$$

The formula above implies that the `double` `4.0 / 3.0` encodes the number

$$1 + \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^8} + \cdots + \frac{1}{2^{52}} = \frac{4}{3} \cdot \left[1 - \frac{1}{2^{54}}\right] \tag{5.7.6.2}$$

and by subtracting 1, this formula implies that the `double` `4.0 / 3.0 - 1.0` encodes the number

$$\frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^8} + \cdots + \frac{1}{2^{52}}. \tag{5.7.6.3}$$

Dividing by $2^2 = 4$, equation (5.7.6.2) also implies that the `double 1.0 / 3.0` encodes the number

$$\frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^8} + \ldots + \frac{1}{2^{52}} + \frac{1}{2^{54}}. \tag{5.7.6.4}$$

Subtracting equation (5.7.6.4) from equation (5.7.6.3) tells us that `4.0 / 3.0 - 1.0 - 1.0 / 3.0` encodes the number $-\frac{1}{2^{54}}$.

If this seems too mathy for you, try calculating $\frac{4}{3} - 1 - \frac{1}{3}$ using decimals, but only allow yourself four significant digits.

- $\frac{4}{3} = 1.333$, so $\frac{4}{3} - 1 = 0.333$;
- $\frac{1}{3} = 0.3333$, so $(\frac{4}{3} - 1) - \frac{1}{3} = 0.333 - 0.3333 = -0.0003 < 0$.

We reach a similar conclusion by doing this analogous calculation. The only difference with the truth is that `double`s use binary instead of decimal and use 53 significant bits instead of 4 significant digits.

In this class, you can often pretend that a `double` is good at representing any real number, but sometimes the truth *will* matter. A `double`'s range and precision is limited and they can only store approximations of most numbers.

Example 5.7.6.5. This code shows a similar example to the one we just looked at but where `20.15` replaces `4.0 / 3.0`. Then we see that `static_cast<int>(100 * 20.15)` is `2014`. This type of issue could mess up a program attempting to convert dollars to pennies. The issue is most easily fixed by using the `round` function.

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

int main() {
    // cout << setprecision(17);
    cout << boolalpha << (20.15 - 20.0 - 0.15 == 0.0) << endl;  // false

    cout <<                                20.15        << endl;  // 20.15
    cout <<                          100 * 20.15        << endl;  // 2015
    cout << static_cast<int>(        100 * 20.15 )      << endl;  // 2014
    cout <<                  round(100 * 20.15)         << endl;  // 2015
    cout << static_cast<int>(round(100 * 20.15))        << endl;  // 2015

    return 0;
}
```

By including `<iomanip>` and writing `cout << setprecision(17)`, we can ask for the output to use 17 significant digits. The first two numerical values would then be displayed as `20.149999999999999` and `2014.9999999999998` which might make what we are observing seem a little more reasonable to you.

5.8 `unsigned int`

An `unsigned int` is similar to an `int`, except that it cannot store negative numbers. An `unsigned int` can store the integers from 0 to $2^{32}-1$ on most computers and compilers. Furthermore, the following is true on most computers and compilers.

Definition 5.8.1. Under the hood, an `unsigned int` consists of 32 bits which are used to encode an integer between 0 and $2^{32}-1$. If $b_0, b_1, b_2, \ldots, b_{29}, b_{30}, b_{31}$ are 32 values, each `0` or `1`, then they encode the integer:

$$b_0 \cdot 2^0 \ + \ b_1 \cdot 2^1 \ + \ b_2 \cdot 2^2 \ + \ b_3 \cdot 2^3 \ + \ \ldots \ + \ b_{28} \cdot 2^{28} \ + \ b_{29} \cdot 2^{29} \ + \ b_{30} \cdot 2^{30} \ + \ b_{31} \cdot 2^{31}$$

Another way to explain this encoding is that each integer between 0 and $2^{32}-1$ is encoded using binary.

Before C++20, casting between `int`s and `unsigned int`s was more complicated than one would hope. Since C++20, the following is true.

Definition 5.8.2. Suppose `i` is an `int` and `u` is an `unsigned int`. We can consider:

- `static_cast<unsigned int>(i)`. This is an `unsigned int`.

  When `i` is non-negative, this casting preserves the value of the stored integer.

  When `i` is negative, this adds 2^{32} the value of the stored integer.

- `static_cast<int>(u)`. This is an `int`.

  When `u` is less than 2^{31}, this casting preserves the value of the stored integer.

  When `i` is greater than or equal to 2^{31}, this subtracts 2^{32} from the value of the stored integer.

Definition 5.8.3. When an arithmetic operation is performed with an `int` and `unsigned int`, the `int` will be cast to an `unsigned int` before the arithmetic is performed.

Example 5.8.4. In the following code, `i + u` is calculated as `static_cast<unsigned int>(i) + u`. Since `i` is negative, this casting adds 2^{32} to the integer value, and the output is `4294967295` because this is $2^{32}-1$.

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

int main() {
    int          i = -1;
    unsigned int u =  0;

    std::cout << i + u << std::endl;

    return 0;
}
```

Although I emphasized the precision issues that `double`s have much more, one should watch out for performing arithmetic with `int`s and `unsigned int`s which would mathematically lead to values which are out of range of the data types.

Example 5.8.5. The following code attempts to calculate $2 \cdot 2^{31}$, but mathematically, this would give 2^{32} which is bigger than the largest `unsigned int`.

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

int main() {
    const unsigned int TWO_TO_THE_THIRTY_ONE = 2147483648;

    std::cout <<     TWO_TO_THE_THIRTY_ONE << "␣";
    std::cout << 2 * TWO_TO_THE_THIRTY_ONE << std::endl;

    return 0;
}
```

The output is `2147483648 0`.

Undefined Behavior 5.8.6. *The aforementioned scenario is called overflow. For `unsigned int`s, it gives defined behavior (calculations modulo 2^{32}), but for `int`s, it gives undefined behavior.*

5.9 `std::size_t`

`size_t`s are like `unsigned int`s, but they use 32 bits on some computers and compilers and 64 bits on other computers and compilers. This data type is used to store...

- *Sizes* of containers.

  We will see `std::string` and `std::vector` as examples of containers.

- *Indices.*

  An index is a number which is used to describe a position in a container.

We will return to this data type as soon as we discuss `std::string`s.

5.10 `char`

The datatype `char` stores its data using exactly one byte. To be completely honest, in rare situations, a byte is not the same as 8 bits. However, in almost all situations, a byte *is* the same as 8 bits, and the `char` datatype can store $2^8 = 256$ different values. In many applications, one only cares about a subset of the 128 values that are included in the ASCII table. The ASCII table is a list of characters and their *codepoints*. A codepoint is simply an integer associated with a specific character. On the following page, I have listed the characters in the ASCII table with codepoints 0, 9, 10, 13, and 32 to 126 using the corresponding `char` literals. Recall that string literals are specified using double quotes. `char` literals are specified using single quotes.

Codepoint	char	Codepoint	char	Codepoint	char	Codepoint	char
0	`'\0'`	32	`' '`	64	`'@'`	96	`` '`' ``
		33	`'!'`	65	`'A'`	97	`'a'`
		34	`'"'`	66	`'B'`	98	`'b'`
		35	`'#'`	67	`'C'`	99	`'c'`
		36	`'$'`	68	`'D'`	100	`'d'`
		37	`'%'`	69	`'E'`	101	`'e'`
		38	`'&'`	70	`'F'`	102	`'f'`
		39	`'\''`	71	`'G'`	103	`'g'`
		40	`'('`	72	`'H'`	104	`'h'`
9	`'\t'`	41	`')'`	73	`'I'`	105	`'i'`
10	`'\n'`	42	`'*'`	74	`'J'`	106	`'j'`
		43	`'+'`	75	`'K'`	107	`'k'`
		44	`','`	76	`'L'`	108	`'l'`
13	`'\r'`	45	`'-'`	77	`'M'`	109	`'m'`
		46	`'.'`	78	`'N'`	110	`'n'`
		47	`'/'`	79	`'O'`	111	`'o'`
		48	`'0'`	80	`'P'`	112	`'p'`
		49	`'1'`	81	`'Q'`	113	`'q'`
		50	`'2'`	82	`'R'`	114	`'r'`
		51	`'3'`	83	`'S'`	115	`'s'`
		52	`'4'`	84	`'T'`	116	`'t'`
		53	`'5'`	85	`'U'`	117	`'u'`
		54	`'6'`	86	`'V'`	118	`'v'`
		55	`'7'`	87	`'W'`	119	`'w'`
		56	`'8'`	88	`'X'`	120	`'x'`
		57	`'9'`	89	`'Y'`	121	`'y'`
		58	`':'`	90	`'Z'`	122	`'z'`
		59	`';'`	91	`'['`	123	`'{'`
		60	`'<'`	92	`'\\'`	124	`'|'`
		61	`'='`	93	`']'`	125	`'}'`
		62	`'>'`	94	`'^'`	126	`'~'`
		63	`'?'`	95	`'_'`		

Definition 5.10.1.

- If `cp` is an `int` storing one of the codepoints listed above, then `static_cast<char>(cp)` produces the `char` with that codepoint.
- If `ch` is a `char` storing one of the `char`s listed above, then `static_cast<int>(ch)` produces the `int` storing the codepoint of that `char`.
- When an arithmetic operation is performed with two `char`s, or a `char` and an `int`, the `char` will be cast to an `int` before the arithmetic is performed. For example, `'A' - 'a'` is calculated as `static_cast<int>('A') - static_cast<int>('a')` which is `65 - 97` which is `-32`.

Example 5.10.2. Since all the lower case alphabetic characters are in order and next to each other and all the upper case alphabetic characters are also in order and next to each other `'A' - 'a'` is the same as `'B' - 'b'` which is the same as `'C' - 'c'` which is the same as... `'Z' - 'z'`. This is the integer that we need to add to a codepoint to go from the codepoint of a lower case letter to the codepoint of the corresponding upper case letter.

In the code below, `ch1 + 'A' - 'a'` casts `ch1` to its codepoint automatically because this expression uses arithmetic, but to cast back to a `char`, we need `static_cast<char>`.

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

int main() {
    cout << static_cast<int>('P') << '␣';  // codepoint of 'P' is 80
    cout << static_cast<int>('I') << '␣';  // codepoint of 'I' is 73
    cout << static_cast<int>('C') << '␣';  // codepoint of 'C' is 67
    cout << static_cast<int>('1') << '␣';  // codepoint of '1' is 49 (not 1)
    cout << static_cast<int>('0') << '␣';  // codepoint of '0' is 48 (not 0)
    cout << static_cast<int>('A') << endl; // codepoint of 'A' is 65

    char ch1 = 'p';
    char ch2 = 'i';
    char ch3 = 'c';
    cout << ch1 << ch2 << ch3 << endl;

    cout << static_cast<char>(ch1 + 'A' - 'a');
    cout << static_cast<char>(ch2 + 'A' - 'a');
    cout << static_cast<char>(ch3 + 'A' - 'a');
    cout << endl;

    return 0;
}
```

The output is ...

```
80 73 67 49 48 65
pic
PIC
```

5.11 `static_cast<T>` and implicit casting

We first spoke about using `static_cast` to cast between `int`s and `double`s. Then we spoke about using it to cast between `int`s and `unsigned int`s, and then for `int`s and `char`s. Hopefully, you can make sensible guesses about how casting works between other pairs of fundamental types, but I should say something about `bool`s.

Definition 5.11.1.

- `static_cast<bool>(0)`, `static_cast<bool>(0.0)`, `static_cast<bool>('\0')` all give `false`.
- If `i` is an `int` not equal to `0`, `d` is a `double` not equal to `0.0`, `c` is a `char` not equal to `'\0'`, then `static_cast<bool>(i)`, `static_cast<bool>(d)`, `static_cast<bool>(c)` all give `true`.
- `static_cast<int>(false)` is equal to `0`,

  `static_cast<double>(false)` is equal to `0.0`,

  `static_cast<char>(false)` is equal to `'\0'`.

  `static_cast<int>(true)` is equal to `1`,

  `static_cast<double>(true)` is equal to `1.0`.

The other thing to mention is that casting occurs automatically and implicitly when assigning an expression of one fundamental type to a variable of another fundamental type. When a casting might be interpreted by someone else as a mistake, I think it is best to be explicit about the casting by using `static_cast`. This will indicate to the other person that the casting is intentional.

Example 5.11.2. In the code below, `8.8` is automatically truncated to `8`; `-1` is implicitly cast to `4294967295`; `'X'` is implicitly cast to `88`; and lots of implicit casting occurs in lines 16 and 22.

```
1  #include <iostream>
2
3  int main() {
4      int i = 8.8;
5      // int i = static_cast<int>(8.8);
6
7      unsigned int u = -1;
8      // unsigned int u = static_cast<unsigned int>(-1);
9
10     int cp = 'X';
11     // int cp = static_cast<int>('X');
12
13     char ch = 'm';
14
15     ch += 'A' - 'a';
16     // ch = ch + 'A' - 'a';
17     // ch = static_cast<int>(ch) + static_cast<int>('A') - static_cast<int>('a');
18     // ch = static_cast<char>(static_cast<int>(ch)
19     //                        + static_cast<int>('A') - static_cast<int>('a'));
20
21     ++ch;
22     // ch = ch + 1;
23     // ch = static_cast<int>(ch) + 1;
24     // ch = static_cast<char>(static_cast<int>(ch) + 1);
25
26     std::cout << i << '␣' << u << '␣' << cp << '␣' << ch << std::endl;
27     return 0;
28 }
```

5.12 Review questions

1. List all problems with the following code, classifying them as one of the following.

   - For all compilers, a build error is encountered.
   - For all compilers, the code successfully builds and encounters a runtime error.
   - There is undefined behavior.

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

int main() {
    const int A_CONSTANT;
    cout << A_CONSTANT << endl;

    return 0;
}
```

2. What happens if a user runs the following code and does not type anything?

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

int main() {
    int i; cin >> i;
    return 0;
}
```

3. Either specify why the following code produces an error or has undefined behavior, or, if there are no problems, write down the output from executing the code.

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

int main() {
    int i = 3;
    int j = 4;
    int k = 20;

    double d1 = (i + j) / 2;
    cout << d1 << endl;

    double d2 = 20.15;
    cout << (d2 - k == 0.15) << endl;

    return 0;
}
```

4. What is the output of the following code?

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

int main() {
    char ch = '2';
    cout <<  boolalpha  << (ch == 2) << endl;
    cout << ('B' - 'a' == 'F' - 'e') << endl;

    return 0;
}
```

6 Strings (std::string)

6.1 Constructing std::strings and using member functions for the first time

We have stored integer values using a variable of type int lots of times now. We have used string literals, but we have not yet stored a sequence of characters in a similar way to storing an integer. The std::string class will solve this problem for us.

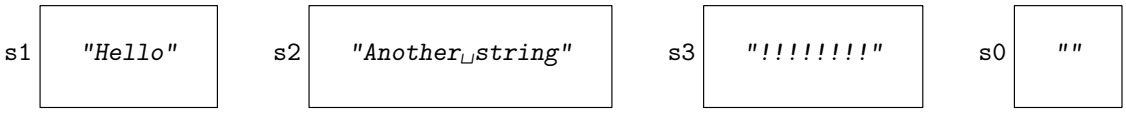
To use the std::string class, we should #include <string>, even if our IDE allows us to omit this; another compiler may not allow this #include to be omitted. Below, four ways of *constructing* a std::string are demonstrated.

- The first looks most similar to how we often initialize an int.
- Once we talk about classes in more detail, we will see that the second way is very common. This line *constructs* the string s2 using the string literal *"Another␣string"*. Yes, string literals are not the same as strings. We will see that strings are much fancier!
- The third way constructs a string using 8 and the char *'!'*. The constructed string contains eight exclamation points.
- Finally, s0 is *default* constructed. The fact that s0 is default constructed means that it has been initialized automatically. The string has no characters and is called the *empty string*.

  We previously saw that using an uninitialized int gives undefined behavior. Using a default constructed string is fine. The difference is that int is a fundamental type and std::string is a class, a difference which we will discuss much more.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   int main() {
6       string s1 = "Hello";
7       string s2("Another␣string");
8       string s3(8, '!');
9       string s0;
10
11      cout << s  << '␣';
12      cout << s2 << '␣';
13      cout << s3 << '␣';
14      cout << s0 << endl;
15
16      return 0;
17  }
```

For now, we will draw the following picture for `s1`, `s2`, `s3`, and `s0`. However, **this picture is a lie**. It has to be: `char`s use one byte each, and so each box in this picture contains a different number of bytes even though they are all boxes for `string`s.



Drawing an accurate picture would take us too far into the future because we would need to know about classes (section 15), pointers (section 17), and heap memory (PIC 10B).

Remark 6.1.1. When commenting on the following code, we have been saying, "`i` is a variable of type `int`." Normally, we will say, "`s` is *an instance of* the `string` class." Again, the reason for the different language is that `string` is a class, not a fundamental type.

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

int main() {
    int i;
    string s;

    return 0;
}
```

Definition 6.1.2. A *class* is a souped-up data type. A class is "souped-up" because it has *member functions*.

Example 6.1.3. The output of the following code is `5 ello ell`. `5` is the number of characters in the word "Hello". Although the reason for the remaining output has not been explained carefully yet, you can make some guesses as to why this is the output. For now, the important part is that `length` and `substr` give our first examples of member functions (of the `string` class).

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

int main() {
    string s = "Hello";

    cout << s.length()     << '␣';
    cout << s.substr(1)    << '␣';
    cout << s.substr(1, 3) << endl;

    return 0;
}
```

Remark 6.1.4. We should make note of the syntax that was used here.

`[instance of a class].[name of the member function](arguments of the member function)`

- First, we wrote `s`.

  In general, we could write any instance of any class.

- After that, we typed `.`

  This "dot" is called the *member access operator*.

- Then we typed `length` or `substr`, the name of the member function being used.

  In general, the member function needs to belong to the class which the instance is an instance of.

- Then inside some parentheses, we listed the arguments for the member function.

  In the case of `length`, there were no arguments, but we still had to write the parentheses.

  In the first use of `substr`, there was one argument: `1`.

  In the second use of `substr`, there were two arguments: `1` and `3`.

Remark 6.1.5. The previous functions we considered — `pow`, `sqrt`, `cos`, `acos`, `round` — are not member functions of a class. Their use does not require the member access operator and they are called *free functions*.

6.2 Indexing

To talk about the individual characters in a string, we can use `operator[]`.

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

int main() {
    string alphabet = "abcdefghijklmnopqrstuvwxyz";

    cout << alphabet[0];
    cout << alphabet[1];
    cout << alphabet[2];
    cout << alphabet[3];
    cout << alphabet[25];
    cout << alphabet[alphabet.length() - 1] << endl;

    return 0;
}
```

The output of the code above is `abcdzz`. This tells us that...

- `'a'` is the `0`-th `char` of `alphabet`;
- `'b'` is the `1`-th `char` of `alphabet`;
- `'c'` is the `2`-th `char` of `alphabet`;
- `'d'` is the `3`-th `char` of `alphabet`;
- `'z'` is the `25`-th `char` of `alphabet`.

In computer science (and set theory), one starts counting at `0`. This can cause a confusing ambiguity for the word "first". Is the first or 1-st `char` of `alphabet` `'a'` or `'b'`? For this reason, I try and say zero-th, one-th, two-th, and three-th even if this sounds a bit weird. I also write `0`-th, `1`-th, `2`-th, and `3`-th, even if this reads weirdly.

`char`	`'a'`	`'b'`	`'c'`	`'d'`	...	`'z'`
index	0	1	2	3	...	25

The numbers describing the positions in a `string` are referred to as *indices*. Due to counting from `0`, the last element in `alphabet` is given by the index `25` element even though there are `26` characters. Therefore, if we need to obtain the last character algorithmically, maybe because a `string` is being constructed using user input, then we need to use `alphabet.length() - 1`, not `alphabet.length()`.

The result of indexing using `operator[]` can be assigned to a `char`. We can make assignments from `char`s, and we can also do all of this with a different member function called `at`...

Example 6.2.1. The output of the following code is `f abcdefghIjklMnopQrstUvwxyz`.

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

int main() {
    string s = "abcdefghijklmnopqrstuvwxyz";

    char fifth_element = s[5];

    s[8] = 'I';
    s[12] += 'A' - 'a';

    s.at(16) = 'Q';
    s.at(20) += 'A' - 'a';

    cout << fifth_element << '␣' << s << endl;

    // s[28];     // undefined behavior
    // s.at(28); // runtime error

    return 0;
}
```

The difference between `operator[]` and `at` is highlighted on lines 18 and 19 of the previous example and by the following comments.

Undefined Behavior 6.2.2.

- *Suppose `s` is a `std::string` and `pos > s.length()`. Typing `s[pos]` gives undefined behavior.*
- *If `s` is a non-`const` `string` and `ch` is not equal to `'\0'`, then `s[s.length()] = ch` gives undefined behavior.*
- *Although accessing `s[s.length()]` does not give undefined behavior, it is usually a mistake.*

Runtime Error 6.2.3. *If `s` is a `std::string` and `pos >= s.length()`, then typing `s.at(pos)` produces a runtime error.*

When making sure your code is correct, a runtime error is much better than undefined behavior. `at` will definitely alert you if you accidentally index *out of range*. The price one pays for this helpful runtime error is a small slowdown, since some checking needs to be done to see whether the index is in range or not.

When storing the size/length of a container (e.g. `string`) or an index for a container, the correct data type is `size_t`. It is good practice to `#include <cstddef>`, but this often comes for free with other headers.

Example 6.2.4. The following code produces an output of `26 abc`.

```
1  #include <iostream>
2  #include <cstddef>
3  #include <string>
4  using namespace std;
5
6  int main() {
7      string s = "abcdefghijklmnopqrstuvwxyz";
8
9      size_t len = s.length();   // storing a length --> size_t
10     cout << len << '␣';
11
12     size_t index = 0;          // storing an index --> size_t
13     cout << s[index++];
14     cout << s[index++];
15     cout << s[index++];
16     cout << endl;
17
18     return 0;
19 }
```

Why are `size_t`s perfect for this? Indices are always positive and indices can be very big because we can have long `string`s like the entire works of Shakespeare. Correspondingly, `size_t`s are always positive and they can store the length of any container.

6.3 Other member functions

`substr` produces substrings using either one or two `size_t`s.

- When only one argument is provided, the substring begins at the specified position, and uses all of the remaining characters in the original string.
  - If the position specified is equal to the string length, the substring is the empty string.
  - If the position specified is greater than the string length, a runtime error follows.
- When two arguments are provided, the first argument means the same as before, but the substring produced will not be longer than the second argument. This means that the substring produced will either have length equal to second argument or it will use all of the remaining characters in the original string, whichever uses fewer characters.

Example 6.3.1.

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

int main() {
    string s = "0123456789";

    cout << s.substr(4)      << endl;     // 456789
    cout << s.substr(4, 4)   << endl;     // 4567
    cout << s.substr(4, 44)  << endl;     // 456789

    cout << s.substr(10)     << endl;     // blank line
    cout << s.substr(10, 4)  << endl;     // blank line

    // cout << s.substr(11)     << endl; // runtime error
    // cout << s.substr(11, 4)  << endl; // runtime error

    return 0;
}
```

`find`, `rfind`, `operator+`, `operator+=`, `push_back`, and `pop_back` are described in the video `8b - string2`. The descriptions available here are also useful.
When `find` does not find what it is looking for the value `static_cast<size_t>(-1)` is returned.

6.4 Review questions

1. Does the following code encounter a build error, a runtime error, or undefined behavior, or does it build and execute without issue on all compilers?

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

int main() {
    string s;
    cout << s << endl;
    return 0;
}
```

2. The following code intends to print the characters in the declared string in reverse? Are there any problems with the code?

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

int main() {
    string s("quiz");
    cout << s[4] << s[3] << s[2] << s[1] << endl;
    return 0;
}
```

3. Does the following code encounter a build error, a runtime error, or undefined behavior, or does it build and execute without issue on all compilers?

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

int main() {
    string s = "abcdef";
    cout << s[8] << endl;
    return 0;
}
```

4. Does the following code encounter a build error, a runtime error, or undefined behavior, or does it build and execute without issue on all compilers?

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

int main() {
    string s = "abcdef";
    cout << s.at(8) << endl;
    return 0;
}
```

5. Which do you intend to use on homework assignments, `at` or `operator[]`? Why?

7 The input buffer

7.1 Motivating the necessity of a deeper understanding of `std::cin`

Receiving user input correctly can be surprisingly difficult and even some examples that you would hope to be very simple require a good understanding of *the input buffer*.

Let's start with two examples that hopefully feel intuitive. If they do not feel intuitive, that is okay because to fully understand these examples requires a more careful explanation of `std::cin` than has been given so far. Right now, I am only trying to motivate why a more careful explanation is necessary.

Example 7.1.1. In this example, the program asks a user for their first and last name using two questions, and then confirms how it recorded the two names.

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

int main() {
    string first_name, last_name;

    cout << "What␣is␣your␣first␣name?␣"; cin >> first_name;
    cout << "What␣is␣your␣last␣name?␣"; cin >> last_name;

    cout << "I␣have␣you␣down␣as␣";
    cout << first_name << '␣' << last_name << '.' << endl;

    return 0;
}
```

If a user runs the program, types `Michael`, presses the ENTER key, types `Andrews`, and presses the ENTER key again, then the console displays...

```
What is your first name? Michael
What is your last name? Andrews
I have you down as Michael Andrews.
```

Example 7.1.2. In this example, the program asks a user for their first and last name using just one question, and then confirms how it recorded the two names.

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

int main() {
    string first_name, last_name;

    cout << "What␣is␣your␣first␣and␣last␣name?␣";
    cin >> first_name >> last_name;

    cout << "I␣have␣you␣down␣as␣";
    cout << first_name << '␣' << last_name << '.' << endl;

    return 0;
}
```

If a user runs the program, types `Michael Andrews` and presses ENTER, then the console displays...

```
What is your first and last name? Michael Andrews
I have you down as Michael Andrews.
```

If we want a program to ask for someone's full name, we have a problem to overcome: people can have a different number of names. Some people only have one name, for example Cher and Ye. Some people have longer names, e.g. Ronaldo Luís Nazário de Lima and Kiefer William Frederick Dempsey George Rufus Sutherland. We can store the user's full name using a single `string` called `full_name`, but the previous example shows that the spaces separating the names will cause `cin >>` to only assign their first name to `full_name`. For this reason, a function called `getline` is useful.

Example 7.1.3. In this example, the program asks a user for their full name and then confirms how it recorded their name.

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

int main() {
    cout << "What␣is␣your␣full␣name?␣";
    string full_name; getline(cin, full_name);

    cout << "I␣have␣you␣down␣as␣";
    cout << full_name << '.' << endl;

    return 0;
}
```

If a user runs the program, types `Kiefer William Frederick Dempsey George Rufus Sutherland` and presses ENTER, the console displays...

```
What is your full name? Kiefer William Frederick Dempsey George Rufus Sutherland
I have you down as Kiefer William Frederick Dempsey George Rufus Sutherland.
```

`getline` solved a problem, but the way `cin >> -` and `getline(cin, -)` interact creates a new one...

Example 7.1.4. The following code is a **failed** attempt to write a program that asks a user for their favorite integer, their full name, and then says something to them using their full name.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      cout << "What␣is␣your␣favorite␣integer?␣";
7      int i; cin >> i;
8
9      cout << "What␣is␣your␣full␣name?␣";
10     string full_name; getline(cin, full_name);
11
12     cout << "I␣like␣you,␣" << full_name << ".␣";
13     cout << i << "␣is␣my␣favorite␣too!" << endl;
14
15     return 0;
16 }
```

If a user runs the program, types `8` and presses ENTER, the program finishes without giving the user an opportunity to type their name and the console displays...

```
What is your favorite integer? 8
What is your full name? I like you, . 8 is my favorite too!
```

This can be fixed by typing `cin.ignore()` at the end of line 7.

Example 7.1.5. The following code is a successful attempt to write a program that asks a user for their favorite integer, their full name, and then says something to them using their full name.

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

int main() {
    cout << "What␣is␣your␣favorite␣integer?␣";
    int i; cin >> i; cin.ignore();

    cout << "What␣is␣your␣full␣name?␣";
    string full_name; getline(cin, full_name);

    cout << "I␣like␣you,␣" << full_name << ".␣";
    cout << i << "␣is␣my␣favorite␣too!" << endl;

    return 0;
}
```

If a user runs the program, types `8`, presses ENTER, types `Tomas Nils Haake`, and presses ENTER, the console displays...

```
What is your favorite integer? 8
What is your full name? Tomas Nils Haake
I like you, Tomas Nils Haake. 8 is my favorite too!
```

To understand this program (which you may have hoped to be less involved) and the previous examples, it is necessary to understand...

- `cin >> variable;`
- `getline(cin, str);`
- `cin.ignore();`

The next subsection is dedicated to an explanation of these function calls.

7.2 `cin >> variable`, `getline(cin, str)`, `cin.ignore()`

Before describing what `cin >> variable`, `getline(cin, str)`, and `cin.ignore()` do, it is necessary to mention the *input buffer*.

When a user of a program is allowed to type into the console and they type something and then press ENTER, the characters they type, including the newline character `\n`, go to an intermediate place called the "input buffer".

- Programs start with an empty input buffer.
- After users have entered input and pressed ENTER, a program can make use of the typed characters via the input buffer.
- A program may or may not use all of the characters in the input buffer by the time it finishes execution.

Example 7.2.1. Consider the following code.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i;
6      cin >> i;
7      return 0;
8  }
```

When the program starts, the input buffer contains no characters at all, but when line 6 is reached, the user is allowed to type input. Let's suppose that the user types `123abcdefghijklmnopqrstuvwxyz` and presses ENTER. The exact characters that the user types go into the input buffer, and so it contains...

```
1  123abcdefghijklmnopqrstuvwxyz\n
```

Notice the newline character at the end. This is because the user pressed ENTER.

In this case, `cin >> i` extracts the characters `123` from the buffer, interprets them as one hundred and twenty-three, and this value is assigned to `i`. At the moment the program ends the following characters are still left within the input buffer.

```
1  abcdefghijklmnopqrstuvwxyz\n
```

Now we are ready to describe `cin >> variable`, `getline(cin, str)`, and `cin.ignore()`.

Definition 7.2.2. Suppose that a variable called `variable` has been declared and defined.
Here are the instructions that `cin >> variable;` follows (when `cin` is not in a fail state)...

1. If there are no characters in the input buffer, allow the user to enter input.
2. Remove all whitespace (spaces, newlines, tabs, `\r`, `\v`, `\f`) from the start of the input buffer.
3. Repeat steps 1 and 2 until non-whitespace is encountered.

   Only when non-whitespace is encountered, move on to step 4.
4. Starting at the beginning of the input buffer, without extracting any whitespace, interpret as many characters as possible as a value of the same type as the type of `variable`, assign this value to `variable`, and strip the interpreted characters from the buffer.

   - When `variable` has type `int` and the input buffer contains `888cat\n`,
     `888` is assigned to `variable` and the buffer is left containing `cat\n`.
   - When `variable` has type `int` and the input buffer contains `888.123\n`,
     `888` is assigned to `variable` and the buffer is left containing `.123\n`.
   - When `variable` has type `double` and the input buffer contains `888.123\n`,
     `888.123` is assigned to `variable` and the buffer is left containing `\n`.
   - When `variable` has type `char` and the input buffer contains `888.123\n`,
     `'8'` is assigned to `variable` and the buffer is left containing `88.123\n`.
   - When `variable` has type `string` and the input buffer contains `Michael A\n`, `"Michael"` is assigned to `variable` and the buffer is left containing the characters in the following string literal `"␣A\n"`. This is because this step should not extract any whitespace.

   It is possible for this step to fail when none of the characters at the start of the input buffer can be interpreted as a value of the same type as the type of `variable`. For example, nothing beginning with the character `a` can be interpreted as a `double`, so when `variable` has type `double` and the input buffer contains `a\n`, this step fails. This causes `cin` to enter a fail state.

Definition 7.2.3. Suppose that a `string` called `str` has been constructed.
Here are the instructions that `getline(cin, str);` follows (when `cin` is not in a fail state)...

1. If there are no characters in the input buffer, allow the user to enter input.
2. Extract everything in the input buffer before `'\n'` and assign it to `str`.

   When the input buffer only contains `\n`, the empty string is assigned to `str`.
3. Extract `'\n'` from the input buffer.

Definition 7.2.4. The instructions that `cin.ignore();` follows (when `cin` is not in a fail state)...

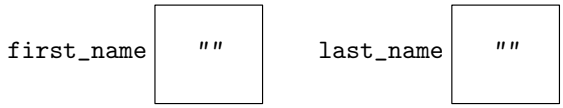
1. If there are no characters in the input buffer, allow the user to enter input.
2. Extract the first character from the input buffer.

7.3 Explanation of motivating examples

We can now explain how the examples in 7.1 execute.

Example 7.3.1. Lets explain how example 7.1.1 executes.

- On line 6 two `string`s are default constructed.

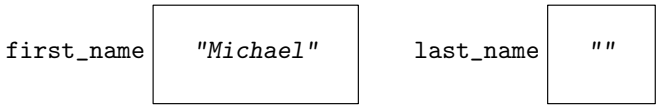


- Line 8 first causes some output to the console so that it looks as follows.

```
1   What is your first name?
```

  To understand what `cin >> first_name;` accomplishes, we look at definition 7.2.2.

  - Because there is nothing in the input buffer, the user is allowed to type.
  - We assumed in example 7.1.1 that they start by typing `Michael` and hitting ENTER.
  - The input buffer now contains `Michael\n`.
  - Since the character `M` is not whitespace, step 2 is completed by doing nothing, and step 3 simply says to go onto step 4.
  - Step 4 says that *`"Michael"`* is assigned to `first_name`, and that the input buffer is left containing `\n` (because whitespace is not extracted at this step).



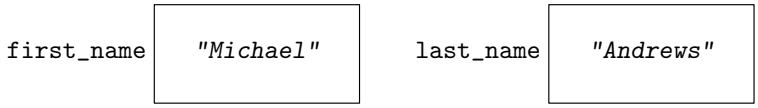
- Line 9 first causes some more output to the console so that it looks as follows.

```
1   What is your first name? Michael
2   What is your last name?
```

  To understand what `cin >> last_name;` accomplishes, we look at definition 7.2.2 again.

  - Because there is still `\n` in the input buffer, **the user is not allowed to type yet.**
  - Step 2 says to remove `\n` from the input buffer which empties it.
  - Step 3 says to go back to step 1.

- Because there is nothing in the input buffer, **the user is now allowed to type.**
- We assumed in example 7.1.1 that they continue by typing `Andrews` and then hit ENTER.
- The input buffer now contains `Andrews\n`.
- Since the character `A` is not whitespace, step 2 is completed by doing nothing, and step 3 simply says to go onto step 4.
- Step 4 says that *`"Andrews"`* is assigned to `last_name`, and the input buffer is left containing `\n` (because whitespace is not extracted at this step).



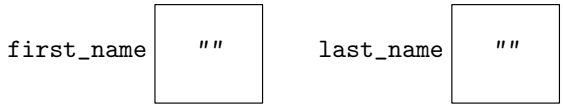
- Line 11 causes some more output as does line 12. The console says.

```
1   What is your first name? Michael
2   What is your last name? Andrews
3   I have you down as Michael Andrews.
```

- `\n` is still in the input buffer as we reach `return 0` and the program finishes executing.

Example 7.3.2. Lets explain how example 7.1.2 executes.

- On line 6 two `string`s are default constructed.

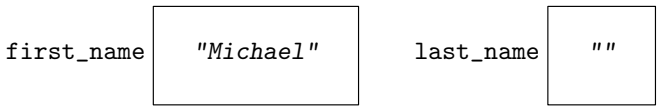


- Line 8 causes some output to the console so that it looks as follows.

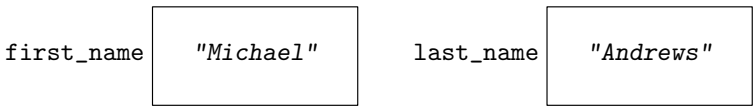
```
1   What is your first and last name?
```

- To understand what `cin >> first_name >> last_name;` accomplishes, we first note that it behaves exactly like two separate `cin` statements: `cin >> first_name; cin >> last_name;`. Therefore, we use definition 7.2.2 twice.

  1. `cin >> first_name;`
     - Because there is nothing in the input buffer, the user is allowed to type.
     - We assumed in example 7.1.2 that they type `Michael Andrews` and then hit ENTER.
     - The input buffer now contains the contents of the string literal *"Michael␣Andrews\n"*.
     - Since the character `M` is not whitespace, step 2 is completed by doing nothing, and step 3 simply says to go onto step 4.
     - Step 4 says that *"Michael"* is assigned to `first_name`, and that the input buffer is left containing the contents of the string literal *"␣Andrews\n"* (because whitespace is not extracted at this step).



  2. `cin >> last_name;`
     - **Because there are still the contents of the string literal *"␣Andrews\n"* in the input buffer, the user is not allowed to type.**
     - Step 2 says to remove the leading space from the input buffer which leaves it containing `Andrews\n`.
     - Since the character `A` is not whitespace, step 3 says to go onto step 4.
     - Step 4 says that *"Andrews"* is assigned to `last_name`, and the input buffer is left containing `\n` (because whitespace is not extracted at this step).



- Line 11 causes some more output as does line 12. The console says.

```
1   What is your first and last name? Michael Andrews
2   I have you down as Michael Andrews.
```

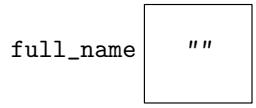
- `\n` is still in the input buffer as we reach `return 0` and the program finishes executing.

Example 7.3.3. Lets explain how example 7.1.3 executes.

- Line 6 causes some output to the console so that it looks as follows.

```
1   What is your full name?
```

- On line 7 a `string` is default constructed.



  To understand what `getline(cin, full_name);` accomplishes, we look at definition 7.2.3.

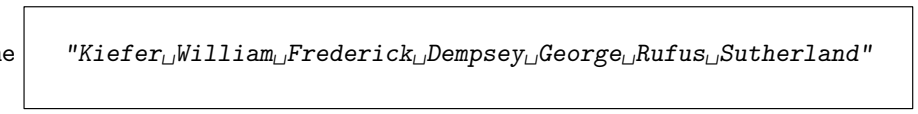
  - Because there is nothing in the input buffer, the user is allowed to type.
  - We assumed in example 7.1.3 that they type `Kiefer William Frederick Dempsey George Rufus Sutherland` and then hit ENTER.
  - The input buffer now contains the contents of the following string literal.

```
1   "Kiefer␣William␣Frederick␣Dempsey␣George␣Rufus␣Sutherland\n"
```

  - Step 2 of definition 7.2.3 says that

```
1   "Kiefer␣William␣Frederick␣Dempsey␣George␣Rufus␣Sutherland"
```

    is assigned to `full_name`...



    ...and that these characters are extracted from the input buffer.

  - Step 3 says that `\n` is extracted from the input buffer which leaves it empty.

- Line 9 and 10 execute to produce the remainder of the output.

- The input buffer is empty as we reach `return 0` and the program finishes executing.

Example 7.3.4. Lets explain how example 7.1.4 executes.

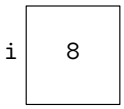
- Line 6 causes some output to the console so that it looks as follows.

```
1   What is your favorite integer?
```

- On line 7 an `int` is declared and defined.

To understand what `cin >> i;` accomplishes, we look at definition 7.2.2.

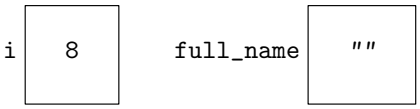
  - Because there is nothing in the input buffer, the user is allowed to type.
  - We assumed in example 7.1.4 that they type `8` and then hit ENTER.
  - The input buffer now contains `8\n`.
  - Since the character `8` is not whitespace, step 2 is completed by doing nothing, and step 3 simply says to go onto step 4.
  - Step 4 says that the `int` `8` is assigned to `i`, and that the input buffer is left containing `\n`.



- Line 9 causes some output to the console so that it looks as follows.

```
1   What is your favorite integer? 8
2   What is your full name?
```

- On line 10 a `string` is default constructed.



To understand what `getline(cin, full_name);` accomplishes, we look at definition 7.2.3.

  - **Because there is still `\n` in the input buffer, the user is not allowed to type.**
  - There are no characters before `\n`, and so step 2 of definition 7.2.3 says that the empty string is assigned to `full_name` leaving the picture as above.
  - Step 3 says that `\n` is extracted from the input buffer which leaves it empty.

- Line 12 causes some more output as does line 13. The console says.

```
1   What is your favorite integer? 8
2   What is your full name? I like you, . 8 is my favorite too!
```

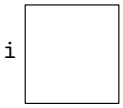
- The input buffer is empty as we reach `return 0` and the program finishes executing.

Example 7.3.5. Lets explain how example 7.1.5 executes.

- Line 6 causes some output to the console so that it looks as follows.

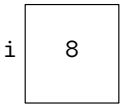
```
1   What is your favorite integer?
```

- On line 7 an `int` is declared and defined.



To understand what `cin >> i;` accomplishes, we look at definition 7.2.2.

  - Because there is nothing in the input buffer, the user is allowed to type.
  - We assumed in example 7.1.5 that they start by typing `8` and hitting ENTER.
  - The input buffer now contains `8\n`.
  - Since the character `8` is not whitespace, step 2 is completed by doing nothing, and step 3 simply says to go onto step 4.
  - Step 4 says that the `int 8` is assigned to `i`, and that the input buffer is left containing `\n`.



Now `cin.ignore();` executes.

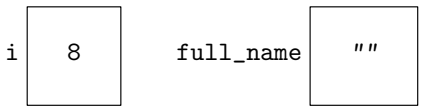
To understand what it accomplishes, we look at definition 7.2.4.

- Because there is still `\n` in the input buffer, the user is not allowed to type.
- The `\n` is the first and only character in the input buffer and it is extracted. **The input buffer is now empty.**

- Line 9 causes some output to the console so that it looks as follows.

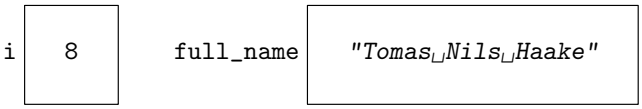
```
1  What is your favorite integer? 8
2  What is your full name?
```

- On line 10 a `string` is default constructed.



To understand what `getline(cin, full_name);` accomplishes, we look at definition 7.2.3.

- **Because there is nothing in the input buffer, the user is allowed to type.**
- We assumed in example 7.1.5 that they type `Tomas Nils Haake` and then hit ENTER.
- The input buffer now contains the contents of the string literal *`"Tomas␣Nils␣Haake\n"`*.
- Step 2 of definition 7.2.3 says that *`"Tomas␣Nils␣Haake"`* is assigned to `full_name`



  and that these characters are extracted from the input buffer.
- Step 3 says that `\n` is extracted from the input buffer which leaves it empty.

- Line 12 causes some more output as does line 13. The console says.

```
1  What is your favorite integer? 8
2  What is your full name? Tomas Nils Haake
3  I like you, Tomas Nils Haake. 8 is my favorite too!
```

- The input buffer is empty as we reach `return 0` and the program finishes executing.

7.4 `cin.get()`, `cin.peek()`, `cin.fail()`, `cin.clear()`

Here are definitions of some other member functions of the `std::istream` class, member functions that `std::cin` can use. We may use some of them later.

Definition 7.4.1. Suppose that a `char` called `ch` has been declared and defined.
Here are the instructions that `ch = cin.get();` follows (when `cin` is not in a fail state)...

1. If there are no characters in the input buffer, allow the user to enter input.
2. Extract the first character from the input buffer and assign it to `ch`.

Definition 7.4.2. Suppose that a `char` called `ch` has been declared and defined.
Here are the instructions that `ch = cin.peek();` follows (when `cin` is not in a fail state)...

1. If there are no characters in the input buffer, allow the user to enter input.
2. Assign the first character in the input buffer to `ch` and leave it in the input buffer, that is, do not extract it from the input buffer.

Definition 7.4.3.

- `cin.fail()` returns `false` when `cin` is not in a fail state.
- `cin.fail()` returns `true` when `cin` is in a fail state.

Definition 7.4.4. `cin.reset()` resets the fail state of `cin` to be `false`.

7.5 Review questions

1. Someone executes the code below and types the following into the console.

```
98   87
65 4321
9 8 765 43 21
```

   To clarify, they hit ENTER at the end of each line immediately after the final characters (`'7'`, `'1'`, and `'1'`, respectively). What is the output produced by the `cout` statements?

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

int main() {
    int    i1, i2, i3, i4, i5;
    char   c;
    string s;

    cin >> i1 >> i2;
    getline(cin, s);

    cin >> i3;
    cin.ignore();
    cin.ignore();

    cin >> c;
    cin >> i4 >> i5;

    cout << "Line␣1:␣" << i1 << endl;
    cout << "Line␣2:␣" << i2 << endl; // These variables
    cout << "Line␣3:␣" << s  << endl; // are printed in
    cout << "Line␣4:␣" << i3 << endl; // the same order
    cout << "Line␣5:␣" << c  << endl; // that they are
    cout << "Line␣6:␣" << i4 << endl; // assigned to.
    cout << "Line␣7:␣" << i5 << endl;

    return 0;
}
```

2. Someone executes the code below and types the following into the console.

```
98   87
65 4321
9 8 765 43 21
```

To clarify, they hit ENTER at the end of each line immediately after the final characters (`'7'`, `'1'`, and `'1'`, respectively). What is the output produced by the `cout` statements?

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

int main() {
    int    i1, i2, i3, i4, i5;
    char   c;
    string s;

    cin >> i1 >> i2;

    cin.ignore();
    getline(cin, s);

    cin >> i3;
    cin.ignore();
    cin.ignore();

    cin >> c;
    cin >> i4 >> i5;

    cout << "Line␣1:␣" << i1 << endl;
    cout << "Line␣2:␣" << i2 << endl; // These variables
    cout << "Line␣3:␣" << s  << endl; // are printed in
    cout << "Line␣4:␣" << i3 << endl; // the same order
    cout << "Line␣5:␣" << c  << endl; // that they are
    cout << "Line␣6:␣" << i4 << endl; // assigned to.
    cout << "Line␣7:␣" << i5 << endl;

    return 0;
}
```

3. The following code builds and runs without error when the user types `xyz` followed by the ENTER key. What does the `cout` statement print to the console?

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

int main() {
    char ch1 = cin.get();
    char ch2 = cin.peek();
    char ch3 = cin.get();

    cout << ch1 << '␣' << ch2 << '␣' << ch3 << endl;

    return 0;
}
```

8 Control Flow

Before this section, when we wrote code, as long as it built and ran without encountering a runtime error, every line executed. Often, however, we will want more control over which lines of our code execute and how many times they execute. Control flow will give us this superpower!

8.1 If

To motivate talking about `if` statements, let's consider an example. Suppose that we want to write a program that asks a user for an `int`, and then prints its absolute value. The absolute value of a number ignores the minus sign: the absolute value of `28` is `28` and the absolute value of `-118` is `118`. What the program does depends heavily on whether the user types a positive or negative number.

This first piece of code behaves correctly when the user types a positive number, but incorrectly when the user types a negative number.

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

int main() {
    cout << "Give␣me␣an␣int␣and␣I␣will␣";
    cout << "tell␣you␣its␣absolute␣value:␣";

    int i; cin >> i;

    cout << "The␣absolute␣value␣of␣" << i << "␣is␣";

    cout << i << endl;
    return 0;
}
```

This second piece of code behaves correctly when the user types negative numbers (because two minus signs cancel), but incorrectly when the user types positive numbers.

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

int main() {
    cout << "Give␣me␣an␣int␣and␣I␣will␣";
    cout << "tell␣you␣its␣absolute␣value:␣";

    int i; cin >> i;

    cout << "The␣absolute␣value␣of␣" << i << "␣is␣";

    cout << -i << endl;
    return 0;
}
```

The only difference between the two pieces of code is found on line 12 and we would like a way to choose between these two lines of code depending on the situation. An `if-else` statement saves the day!

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout << "Give␣me␣an␣int␣and␣I␣will␣";
6      cout << "tell␣you␣its␣absolute␣value:␣";
7
8      int i; cin >> i;
9
10     cout << "The␣absolute␣value␣of␣" << i << "␣is␣";
11
12     if (i >= 0) {
13         cout <<  i << endl;
14     }
15     else {
16         cout << -i << endl;
17     }
18     return 0;
19 }
```

When `i >= 0` evaluates to `true`, line 13 executes. When `i >= 0` evaluates to `false`, line 16 executes. An `if-else` statement allows us to choose between executing two different blocks of code and the expression typed within the parentheses determines which block of code executes.

The previous example was naturally solved using an `if-else` statement, but an `if` statement can be used without an associated `else` clause.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout << "Give␣me␣an␣int␣and␣I␣will␣tell␣you␣its␣parity:␣";
6      int i; cin >> i; cout << i << "␣is␣";
7
8      if (i % 2 == 0) {
9          cout << "even" << endl;
10     }
11     else {
12         cout << "odd" << endl;
13     }
14
15     if (i == 8) {
16         cout << "Also,␣you␣happened␣to␣pick␣my␣favorite␣number!" << endl;
17     }
18     return 0;
19 }
```

The code above gives another example of an `if-else` statement on lines 8 to 13. The `if` statement on lines 15 to 17 does not have a corresponding `else` clause. The program informs a user who picks `8` that it is their favorite number. It does not share this information with those who do not pick `8`, and it does not say anything else to these users in place of this information: when `i == 8` evaluates to `false`, line 16 does not execute and no other line of code executes in its place.

Definition 8.1.1. An `if` statement uses the following the following syntax.

```
1  if (condition) {
2      statements
3  }
```

It must be possible for the condition to be evaluated to give a `bool`. When the condition evaluates to `true`, the statements within the braces `{}` are executed. When the condition evaluates to `false`, the statements within the braces `{}` are not executed.

Definition 8.1.2. An `if-else` statement uses the following the following syntax.

```
1  if (condition) {
2      if-statements
3  }
4  else {
5      else-statements
6  }
```

It must be possible for the condition to be evaluated to give a `bool`. When the condition evaluates to `true`, the if-statements are executed and the else-statements are not executed. When the condition evaluates to `false`, the if-statements are not executed and the else-statements are executed.

Definition 8.1.3. `else if` provides further useful syntax. It is possible to abbreviate...

```
1  if (condition_1) {
2      statements_1
3  }
4  else {
5      if (condition_2) {
6          statements_2
7      }
8      else {
9          statements_3
10     }
11 }
```

as...

```
if (condition_1) {
    statements_1
}
else if (condition_2) {
    statements_2
}
else {
    statements_3
}
```

...and it is possible to abbreviate...

```
if (condition_1) {
    statements_1
}
else {
    if (condition_2) {
        statements_2
    }
    else {
        if (condition_3) {
            statements_3
        }
        else {
            statements_4
        }
    }
}
```

as...

```
if (condition_1) {
    statements_1
}
else if (condition_2) {
    statements_2
}
else if (condition_3) {
    statements_3
}
else {
    statements_4
}
```

...and this pattern continues to an arbitrary number of conditions and statements.

Furthermore, it is not necessary to end with an `else`. It is fine to end with an `else if`.

Example 8.1.4. In the United States, it is often legal to turn right through a red traffic light. This differs from the United Kingdom and can allow one to arrive at their destination quicker than expected. Unfortunately, many LA drivers don't know how to push their right foot down onto their accelerator and so progress can still be slower than desired. For these reasons, a frustrated driver wrote the following program.

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

int main() {
    cout << "This␣program␣teaches␣LA␣drivers␣" << endl;
    cout << "how␣to␣interpret␣traffic␣lights␣" << endl;
    cout << "when␣stopped␣and␣turning␣right.␣" << endl;
    cout << endl;

    char response;
    bool things_to_hit;
    bool green_light;
    bool no_right_turn_on_red;

    cout << "Are␣there␣pedestrians␣or␣vehicles␣you␣will␣hit?␣(y/n)?␣";
    cin >> response; things_to_hit = (response != 'n');

    cout << "Is␣the␣light␣green␣(y/n)?␣";
    cin >> response; green_light = (response == 'y');

    cout << "Does␣it␣say␣\"no␣right␣turn␣on␣red\"␣(y/n)?␣";
    cin >> response; no_right_turn_on_red = (response != 'n');

    if (things_to_hit) {
        cout << "STAY␣STOPPED!" << endl;
    }
    else {
        if (green_light) {
            cout << "GO!" << endl;
        }
        else {
            if (no_right_turn_on_red) {
                cout << "STAY␣STOPPED!" << endl;
            }
            else {
                cout << "GO!" << endl;
            }
        }
    }

    return 0;
}
```

The code above uses both `response != 'n'` and `response == 'y'` (`!=` is the "not equals" comparison operator). Why? This way lines 16, 19, and 22 make the least dangerous assumptions when someone types characters other than `'y'` and `'n'`. For example, if someone types `'t'` in response to "Are there pedestrians or vehicles you will hit?", the program will behave as though their response was `'y'` (both `'t'` and `'y'` do not equal `'n'`). It is better not to hit pedestrians! Similarly, it will assume the light is red unless the user enters `'y'`, and it will assume that a right turn on red is not allowed unless the user clarifies there is no such sign with a perfect response of `'n'`.

The code above can be rewritten to use `else if` as follows.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       cout << "This␣program␣teaches␣LA␣drivers␣" << endl;
6       cout << "how␣to␣interpret␣traffic␣lights␣" << endl;
7       cout << "when␣stopped␣and␣turning␣right.␣" << endl;
8       cout << endl;
9
10      char response;
11      bool things_to_hit;
12      bool green_light;
13      bool no_right_turn_on_red;
14
15      cout << "Are␣there␣pedestrians␣or␣vehicles␣you␣will␣hit?␣(y/n)?␣";
16      cin >> response; things_to_hit = (response != 'n');
17
18      cout << "Is␣the␣light␣green␣(y/n)?␣";
19      cin >> response; green_light = (response == 'y');
20
21      cout << "Does␣it␣say␣\"no␣right␣turn␣on␣red\"␣(y/n)?␣";
22      cin >> response; no_right_turn_on_red = (response != 'n');
23
24      if (things_to_hit) {
25          cout << "STAY␣STOPPED!" << endl;
26      }
27      else if (green_light) {
28          cout << "GO!" << endl;
29      }
30      else if (no_right_turn_on_red) {
31          cout << "STAY␣STOPPED!" << endl;
32      }
33      else {
34          cout << "GO!" << endl;
35      }
36
37      return 0;
38  }
```

Example 8.1.5. The wikipedia page for Gregorian calendar specifies the rule for a leap year as...

> Every year that is exactly divisible by four is a leap year,
> except for years that are exactly divisible by 100,
> but these centurial years are leap years if they are exactly divisible by 400.
> For example,
> the years 1700, 1800, and 1900 are not leap years,
> but the year 2000 is.

The following code uses this rule to tell a user when an entered year is a leap year.

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

int main() {
    cout << "Type␣a␣year␣and␣I␣will␣tell␣you␣";
    cout << "whether␣it␣is␣a␣leap␣year␣or␣not:␣";

    int year; cin >> year; cout << endl;

    if (year % 4 == 0) {
        cout << year << "␣is␣divisible␣by␣4," << endl;

        if (year % 100 == 0) {
            cout << year << "␣is␣divisible␣by␣100," << endl;

            if (year % 400 == 0) {
                cout << year << "␣is␣divisible␣by␣400," << endl;
                cout << "so␣it␣is␣a␣leap␣year."          << endl;
            }
            else {
                cout << year << "␣is␣not␣divisible␣by␣400," << endl;
                cout << "so␣it␣is␣not␣a␣leap␣year."          << endl;
            }
        }
        else {
            cout << year << "␣is␣not␣divisible␣by␣100," << endl;
            cout << "so␣it␣is␣a␣leap␣year."              << endl;
        }
    }
    else {
        cout << year << "␣is␣not␣divisible␣by␣4," << endl;
        cout << "so␣it␣is␣not␣a␣leap␣year."        << endl;
    }
    return 0;
}
```

bools can be combined in a few ways. To understand how they can be combined it is useful to think of a few statements.

- `i` is non-negative (`i >= 0`).
- `i` is non-positive (`i <= 0`).
- `i` is even (`i % 2 == 0`).

We can put any two of these statements together using the word "and". For example, "`i` is non-negative and `i` is even." This statement is `true` exactly when the two separate statements are `true` and you can see this when `i` is `8`. As soon as one part or both fail to be true, the whole statement becomes `false`. For example, when `i` is `7` (the even part fails), when `i` is `-8` (the non-negative part fails), or when `i` is `-7` (both parts fail).

&&	false	true
false	false	false
true	false	true

We can also put any two of the statements above together using the word "or". For example, "`i` is non-negative or `i` is non-positive." This statement sounds `true` because we know every number is non-negative or non-positive. Reading the statement for specific values of `i` can sound a little weird, but what we just noted justifies why a statement involving "or" should be regarded as `true` as long as at least one of the separate statements is `true`. When `i` is `8`, the first first part (non-negative) is `true`. When `i` is `-8`, the second first part (non-positive) is `true`. When `i` is `0`, both parts are `true`. This last observation highlights that "or" is inclusive in math and computer science, whereas many English sentences use it exclusively.

||	false	true
false	false	true
true	true	true

Finally, "not" changes a statement from `true` to `false` and from `false` to `true`.

b	!b
false	true
true	false

In C++, we have operations on bools: `&&` for "and", `||` for "or", and `!` for "not". The relevant "truth tables" have been provided above.

Example 8.1.6. Going back to the leap year example, the following code provides less information to the user about why a given year is a leap year or not, but the conclusion is correct, and the code uses `&&` and `||` nicely to shorten the control flow.

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

int main() {
    cout << "Type␣a␣year␣and␣I␣will␣tell␣you␣";
    cout << "whether␣it␣is␣a␣leap␣year␣or␣not:␣";

    int year; cin >> year;

    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
        cout << year << "␣is␣a␣leap␣year." << endl;
    }
    else {
        cout << year << "␣is␣not␣a␣leap␣year." << endl;
    }
    return 0;
}
```

Example 8.1.7. It is quite common to use an `if`-statement condition to compare two values for equality. It is unsurprising that writing `=` (assignment) instead of `==` (equality) can produce buggy code, but it is good to see how devastating this mistake can be.

The following code builds and executes without encountering a runtime error. However, line 8 does not execute. This is because the expression `i = 0` evaluates to `0` (the value that is assigned to `i`) and `static_cast<bool>(0)` is `false`.

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

int main() {
    int i = 0;

    if (i = 0) {
        cout << "Code␣involving␣cout␣is␣executing" << endl;
    }
    return 0;
}
```

Example 8.1.8. Recall that the `find` member function of the `std::string` class returns a `size_t` indicating where a character is found, or `static_cast<size_t>(-1)` when a character is not found. Moreover, recall that applying `static_cast<bool>` to integer types gives `false` when the integer is `0` and `true` otherwise. For these reasons, the output of the following code is `bc`, and we deduce that the way the code is written is highly misleading.

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

int main() {
    std::string s = "ab";

    if (s.find('a')) { std::cout << 'a'; }
    if (s.find('b')) { std::cout << 'b'; }
    if (s.find('c')) { std::cout << 'c'; }

    std::cout << std::endl;
    return 0;
}
```

Example 8.1.9. Whenever writing code within the braces of an `if` statement, it is good to indent by four spaces. Correctly used indentations make code much easier to read. Conversely, incorrectly used indentations can make code very confusing to read.

In this example, only pseudocode is provided. The first code only does something when a user attempts to login: if they provide the correct credentials, it welcomes them; otherwise, it informs them that they provided incorrect information.

```
if (attempted_login) {
    if (correct_credentials) {
        // welcome user
    }
    else {
        // inform user about incorrect credentials
    }
}
```

The indentations in the second pseudocode create the illusion that it does the same.

```
{
  if (attempted_login) {
    if (correct_credentials) {
        // welcome user
}
    } else {
        // inform user about incorrect credentials
    }
}
```

However, upon closer inspection, we see that it is enclosed in some braces (that only introduce a scope (next subsection)), and that the `else` is paired with the first `if`, not the second `if`. Without a user even attempting to login, the code informs them of incorrect credentials, and if a user provides incorrect credentials, the user is not given any warning that this has happened. This code performs poorly and it is confusing to debug.

8.2 Scope

The following code produces a build error.

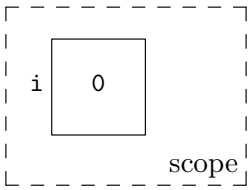
```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      if (true) {
6          int i = 0;
7      }
8      cout << i << endl;
9
10     return 0;
11 }
```

This is because braces `{}` introduce and destroy *scopes*, so that line 8 views `i` as undeclared. It is useful to have a mental image for a scope, so let's look at an example and draw a picture.

Example 8.2.1.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      {
6          int i = 0;
7          cout << i << endl;
8      }
9      // cout << i << endl;
10
11     return 0;
12 }
```

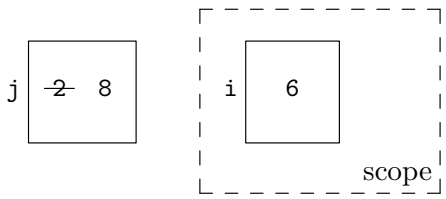
Line 5 introduces a scope, which has been drawn as a dashed box labelled by "scope". On line 6, the `int i` is declared and defined, and its box is drawn inside the scope's box. Line 7 successfully prints `i`'s value of `0`. Line 8 destroys the scope created by line 5 and everything within it. Therefore, by line 9, it is as though `i` was never declared in the first place, and uncommenting the code would produce a build error.



Example 8.2.2. Here is another instructive example.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int j = 2;
6
7       {
8           int i = 3 * j;
9
10          cout << i << '␣';
11
12          j += i;
13      }
14
15      cout << j << endl;
16      // cout << i << endl;
17
18      return 0;
19  }
```

- Line 5 declares and defines an `int` `j` and initializes it with value `2`.
- Line 7 introduces a scope.
- Line 8 declares and defines an `int` `i` and initializes it with value `6`.
  This occurs within the scope introduced by line 7.
- Line 10 prints `i`'s value of `6`.
- Line 12 increments `j`'s value using `i`'s value to change it to `8`.
  The picture drawn below is of the code execution after this step.



- Line 13 destroys the scope introduced by line 7.
- Line 15 prints `j`'s value of `8`.
- By line 16, it is as though `i` was never declared in the first place, and uncommenting the code would produce a build error.
- In summary, the code output is `6 8`, and the `i` introduced on line 8 had a lasting impact on `j`'s value even after it went "out of scope".

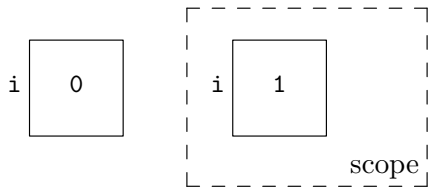
Example 8.2.3. Here is another instructive example.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int i = 0;
6
7       {
8           int i = 1;
9           cout << i << '␣';
10      }
11
12      cout << i << endl;
13
14      return 0;
15  }
```

- Line 5 declares and defines an `int` `i` and initializes it with value `0`.
- Line 7 introduces a scope.
- Line 8 declares and defines another `int` `i` and initializes it with value `1`.

  Because this occurs within the scope introduced by line 7, it is not regarded as a redefinition. There are now two different variables called `i`.

  The picture drawn below is of the code execution after this step.



- Line 9 prints the inner `i`'s value of `1`.
- Line 10 destroys the scope introduced by line 7.
- Line 12 prints the remaining `i`'s value, that is, `0`.
- In summary, the code output is `1 0`, and at one point during the code execution we saw that one `i` *masked* the other `i`.

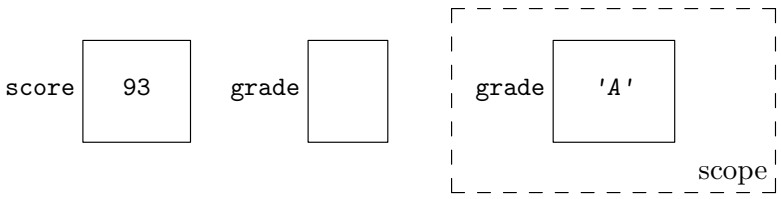
All of this has consequences for code that involves `if`, `if-else`, and/or `else if`. Many new students encounter bugs because they do not pay close enough attention to scopes.

Example 8.2.4. The following code encounters undefined behavior when a user types `93`.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       cout << "Tell␣me␣your␣high␣school␣class␣score␣";
6       cout << "and␣I'll␣tell␣you␣your␣letter␣grade:␣";
7
8       int score; cin >> score;
9       char grade;
10
11      if      (score >= 90) { char grade = 'A'; }
12      else if (score >= 80) {      grade = 'B'; }
13      else if (score >= 70) {      grade = 'C'; }
14      else if (score >= 60) {      grade = 'D'; }
15      else                  {      grade = 'F'; }
16
17      cout << "Your␣grade␣is␣"  << grade << endl;
18
19      return 0;
20  }
```

- Line 8 declares and defines an `int` called `score`.
- `cin >> score` prompts the user for input.

  If they type `93`, then the value of `93` is assigned to `score`.
- Line 9 declares and defines a `char` called `grade`.
- Line 11 executes to introduce a new scope, and another `char` called `grade` within that scope. This variable is initialized with `'A'`.



- The scope introduced on line 11 is destroyed by line 17. Therefore line 17 uses an uninitialized variable and gives undefined behavior.

8.3 While

An `if` statement allows us to specify a condition under which some lines of code execute. The lines within the braces `{}` which follow `if (condition)` will either execute once or not at all. A `while` loop, on the other hand, can allow some lines of code execute to execute many times.

Definition 8.3.1. A `while` loop uses the following the following syntax.

```
while (condition) {
    statements
}
```

It must be possible for the condition to be evaluated to give a `bool` (which, as some earlier examples for `if` statements showed, might involve casting to `bool`).

- When the condition evaluates to `false`, the statements within the braces `{}` are not executed.
- When the condition evaluates to `true`, the statements within the braces `{}` are executed at least once, but they may be executed many more times. This is because the `while` loop above is equivalent to the following.

```
START_OF_WHILE:
{
    if (condition) {
        statements

        goto START_OF_WHILE;
    }
}
```

  Once all of the statements are executed, the condition is checked again. If it evaluates to `true` again, the statements will execute again, and so on.

- If the loop needs to be terminated within the statements, a `break` statement can be used as a terminating statement.

Example 8.3.2. The following code builds and runs without encountering an error.

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

int main() {
    int i = 0;
    const int N = 3;

    while (i < N) {
        cout << "i␣==␣" << i << ",␣";
        cout << "N␣==␣" << N << ",␣";

        cout << "i␣<␣N␣is␣true,␣";
        cout << "so␣the␣body␣of␣the␣'while'␣loop␣executes" << endl;

        ++i;
    }

    cout << "i␣==␣" << i << ",␣";
    cout << "N␣==␣" << N << ",␣";

    cout << "i␣<␣N␣is␣false,␣";
    cout << "so␣the␣we␣have␣left␣the␣body␣of␣the␣'while'␣loop" << endl;

    return 0;
}
```

The output is

```
i == 0, N == 3, i < N is true, so the body of the 'while' loop executes
i == 1, N == 3, i < N is true, so the body of the 'while' loop executes
i == 2, N == 3, i < N is true, so the body of the 'while' loop executes
i == 3, N == 3, i < N is false, so the we have left the body of the 'while' loop
```

Soon enough, we will learn about `for` loops, and we will discover that this example is more suitable for a `for` loop. However, it gives a first example of a `while` loop repeating a few lines of code more than once. Each time the lines execute `i` has a different value. In fact, this is really important. `i` increasing ensures that `i < N` eventually becomes `false` which allows the loop to end.

Example 8.3.3. The following code shows an *infinite loop*. `true` will always evaluate to `true`, and so the contents of the braces, nothing, will continue to execute over and over again, and line 3 will never be reached.

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

This is not a build error or a runtime error. In this case, it is also not undefined behavior. Therefore, it is simplest to classify it only as "infinite loop".

Undefined Behavior 8.3.4. *An infinite loop produces undefined behavior if...*

- *the loop is not the trivial infinite loop* `while (true) {}`,
- *the loop has no observable effects, in particular, it does not print to the console.*

The previous example suggests that `while (true)` is very dangerous, but it is not as bad as it seems because we have the `break` keyword. A good way to think about `while (true)` is "keep going until told to `break`".

Example 8.3.5. The following code produces the exact same output as example 8.3.2.

What is within the braces keeps executing until the `break` keyword on line 10 is encountered. Because of the `if` statement, line 10 executes only when `!(i < N)` is `true`, that is, when `(i < N)` is `false`. This example does not really show the benefit of `while (true)`, but at least you can see the `break` statement doing what it is supposed to do.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i = 0;
6      const int N = 3;
7
8      while (true) {
9          if (!(i < N)) {
10             break;
11         }
12         cout << "i␣==␣" << i << ",␣";
13         cout << "N␣==␣" << N << ",␣";
14
15         cout << "i␣<␣N␣is␣true,␣";
16         cout << "so␣the␣body␣of␣the␣'while'␣loop␣executes" << endl;
17
18         ++i;
19     }
20
21     cout << "i␣==␣" << i << ",␣";
22     cout << "N␣==␣" << N << ",␣";
23
24     cout << "i␣<␣N␣is␣false,␣";
25     cout << "so␣the␣we␣have␣left␣the␣body␣of␣the␣'while'␣loop" << endl;
26
27     return 0;
28 }
```

`while` loops are particularly beneficial when you do not know how many times a process will need to repeat.

Example 8.3.6. If the goal is to write a program that prompts a user for a non-negative `int` and uses it for some purpose, then there is a problem to overcome. We do not know how many times the potentially disobedient user will type a negative number. We need to keep reasking them until they finally obey the instructions! This problem is solved elegantly by a `while` loop.

Here are two slightly different solutions.

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

int main() {
    int i = -1;

    while (i < 0) {
        cout << "Type␣a␣non-negative␣int:␣";
        cin >> i;
    }

    cout << "Thanks!␣" << i << "␣is␣non-negative!" << endl;
    return 0;
}
```

I slightly prefer the second solution because it does not require initializing `i` with a negative value. I do not like making choices and I had to choose `-1` from all the negative `int`s in the first solution. The second solution also reads more like "keep asking the user until they respond correctly" rather than "keep asking the user while they respond incorrectly and kind of assume that they answered incorrectly before we even asked them anything".

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

int main() {
    int i;

    while (true) {
        cout << "Type␣a␣non-negative␣integer:␣";
        cin >> i;

        if (i >= 0) { break; }
    }

    cout << "Thanks!␣" << i << "␣is␣non-negative!" << endl;
    return 0;
}
```

In both cases, if the user types `-1\n-23\n-456\n-7890\n28\n`, then the console displays the following.

```
Type a non-negative integer: -1
Type a non-negative integer: -23
Type a non-negative integer: -456
Type a non-negative integer: -7890
Type a non-negative integer: 28
Thanks! 28 is non-negative!
```

8.4 Solving some user-input problems

In this section, we will work on solving a few user-input problems. This will culminate in writing a C++ program that prompts a user for a list of `int`s separated by spaces and then prints the sum of the `int`s that they enter.

Example 8.4.1. It is good to start off simple. This program has the deficiency that it will not end unless the user stops the program manually, for example, by pressing the stop button in XCode, or by closing the console opened by Visual Studio by clicking on the cross to the top-right. However, it demonstrates the utility of a `while (true)` because the program enables a (quite bad) conversation with a user.

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

int main() {
    while (true) {
        cout << "Type␣something␣and␣I␣will␣repeat␣it..." << endl;

        string s; getline(cin, s);
        cout << s << endl;
    }
    return 0;
}
```

Example 8.4.2. It is good to solve the deficiency of the previous program. In the next program we repeatedly ask a user whether they want to continue, indicating that their response should be `y` or `n`. The program ends when the user responds as instructed and with an `n`.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       while (true) {
6           cout << "Continue␣(y␣/␣n)?␣";
7
8           char response = cin.get(); cin.ignore();
9           if (response == 'n') { break; }
10      }
11      return 0;
12  }
```

There are some comments to make about this program.

- If the user starts by typing `y` and hitting ENTER, then the characters `y\n` will go into the input buffer. The `cin.get()` on line 8 will use and extract the `y` and the `cin.ignore()` on line 8 will extract the `\n`.
- Similarly, if the user starts by typing `n` and hitting ENTER, then the characters `n\n` will go into the input buffer. The `cin.get()` on line 8 will use and extract the `n` and the `cin.ignore()` on line 8 will extract the `\n`.
- Therefore, if we remove `cin.ignore()`, the prompts `Continue (y / n)?` will often be printed twice between what the user types because an entire iteration of the `while` loop will be needed to extract each newline character.
- The program is not very robust to bad user input. If the user types the contents of the string literal `"y␣\nn\nn\nnn\n"`, then the program will end with the console confusingly saying...

```
1   Continue (y / n)? y
2   Continue (y / n)? n
3   Continue (y / n)? n
4   Continue (y / n)? nn
5   Continue (y / n)?
```

The space after the `y` messes everything up: `y` is got, space is ignored, `\n` is got, `n` is ignored, `\n` is got, `n` is ignored, `\n` is got, `n` is ignored, `n` is (finally) got, `\n` is ignored.

You could try and write something more robust if you like, for example, by using `cin >>`.

Example 8.4.3. Even though the prompt in the following code asks the user to enter a list of `int`s separated by single spaces, it can handle more than one space between each pair of `int`s. However, it cannot handle a space after the last `int`. This code assumes that no whitespace is ever typed immediately before the ENTER key is pressed, that is, it assumes that the user always types an integer right before they press ENTER.

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

int main() {
    cout << "Enter␣a␣list␣of␣ints␣with␣a␣single␣space␣";
    cout << "between␣each␣adjacent␣pair␣of␣ints..." << endl;

    int sum = 0;

    while (true) {
        int i; cin >> i;
        sum += i;

        char next = cin.get();
        if (next == '\n') { break; }
    }

    cout << "Their␣sum␣is␣" << sum << endl;
    return 0;
}
```

The code contains some useful ideas.

- To calculate a sum in C++ it is often useful to initialize a variable with a value of `0` and keep increasing it by the values that we want to add together.
- It is not necessary to store all the `int`s that a user types. Since the variable `sum` keeps track of a running total, we only need to keep track of the most recent `int` that has been interpreted by `cin >>`.
- Since `cin >> i` extracts whitespace *before* interpreting characters as an `int`, the variable `next` will always be assigned a `char` consisting of whitespace. Since we assumed that the user does not type spaces after their last `int`, `next` will store `\n` exactly when the last `int` has just been processed. This is why it is used as the condition to `break` from the `while` loop.

Example 8.4.4. Personally, I find that I use `do-while` loops very infrequently which is why more attention is not dedicated to them in this course. The only differences with a `while` loop are...

- the statements in their braces are guaranteed to start executing the first time through;
- the condition to keep going is checked at the end of the loop.

The following code is a minor rewrite of the previous example.
This code is still unable to handle a space after the last `int`.

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

int main() {
    cout << "Enter␣a␣list␣of␣ints␣with␣a␣single␣space␣";
    cout << "between␣each␣adjacent␣pair␣of␣ints..." << endl;

    int sum = 0;

    do {
        int i; cin >> i;
        sum += i;
    }
    while (cin.get() != '\n');

    cout << "Their␣sum␣is␣" << sum << endl;
    return 0;
}
```

Example 8.4.5. This version can handle trailing spaces after the last `int`.

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

int main() {
    cout << "Enter␣a␣list␣of␣ints␣separated␣by␣spaces..." << endl;

    int sum = 0;

    while (true) {
        int i; cin >> i;
        sum += i;

        while (cin.peek() == '␣' ) { cin.ignore(); }
        if    (cin.peek() == '\n') { break;        }
    }

    cout << "Their␣sum␣is␣" << sum << endl;
    return 0;
}
```

Since `cin >> i` extracts whitespace *before* interpreting characters as an `int`, the characters at the start of the input buffer will always be whitespace after `cin >> i` has just executed. The trick used here is to `peek` at them and `ignore` them when they are spaces. Once all the spaces are ignored, we peek again, and if we see `\n`, this indicates we have processed the last `int` and should `break` from the `while` loop. The result of the final `peek` will be `\n`, `-`, or a digit character. It is important that we do not `get` this character, since this could ruin one of the integers that the user typed.

All of the summing examples above respond poorly to users who type letter characters. Handling all the ways in which users can be incompetent is often a lengthy process, and not something we will dedicate more time to here!

8.5 For

Suppose that we want to write a program that starts by declaring and defining a `const int N`, and then sums the integers from `1` to `N` (with the convention that the answer is `0` when `N` is less than `1`). The following code correctly claims that the sum of the integers from 1 to 5 is 15.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      const int N = 5;
6      int sum = 0;
7
8      {
9          int i = 1;
10
11         while (i <= N) {
12             sum += i;
13
14             ++i;
15         }
16     }
17
18     cout << "The␣sum␣of␣the␣integers␣from␣1␣to␣" << N;
19     cout << "␣is␣" << sum << endl;
20
21     return 0;
22 }
```

To sum the integers, the code uses an idea we have seen before: initialize a variable with a value of `0` and keep increasing it by the values that we want to add together. To obtain the values we want to add together, a variable called `i` is initialized with the first value, `1`, and we use a `while` loop to repeatedly increase `sum` by the value of `i` and increment `i` to the next value of interest. One could wonder why a scope is introduced and destroyed on lines 8 and 16. Well, even if we were writing a longer program that does other things, it feels like `i` has served its purpose. It allowed us to sum the values from `1` to `N` and by the end it stores the value of `N + 1`. There is no reason that we need a variable storing `N + 1` so we might as well destroy it. This way, if we need a variable called `i` later on, we do not need to remember that one was already defined above; we can declare and define a new one without encountering a build error.

The structure of lines 8 to 16 appears very frequently in code. This is the reason that `for` loops exist: to make it easier to code this type of idea.

Definition 8.5.1. A `for` loop uses the following syntax.

```
1  for (init-statement; condition; expression) {
2      statements
3  }
```

- The `init-statement` is normally one of the following.
  - An assignment like `i = 0`.
  - A declaration, definition, and initialization like `size_t i = 0`.
  - A declaration, definition, and initialization of two variables of the same type like `size_t i = 0, N = s.length()`.
- It must be possible for the `condition` to be evaluated to give a `bool`.

  As before, this might involve implicitly casting to a `bool`.
- The `expression` just before the closing parenthesis is usually an expression that increments a loop counter like `++i` or `i += 2`.
- If the loop needs to be terminated within the statements, a `break` statement can be used as a terminating statement.

The `for` loop above is almost equivalent to the following.

```
{
    init-statement;

    while (condition) {
        statements

        expression;
    }
}
```

Example 8.5.2. Taking `init-statement` to be `int i = 1`, `condition` to be `i <= N`, `statements` to be `sum += i;` and `expression` to be `++i` we can write the motivating example as follows.

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

int main() {
    const int N = 5;
    int sum = 0;

    for (int i = 1; i <= N; ++i) {
        sum += i;
    }

    cout << "The␣sum␣of␣the␣integers␣from␣1␣to␣" << N;
    cout << "␣is␣" << sum << endl;

    return 0;
}
```

This reads quite nicely. It almost says, "for the values `1` to `N` add these values to `sum`".

Remark 8.5.3. The "almost equivalent" in the previous definition is super close to "equivalent".

One difference is that an empty `condition` in a `for` loop behaves like `true`, whereas an empty condition will give a build error in a `while` loop. There are also subtle differences regarding scope that are only likely to matter when trying to do something strange. For example, if you take...

- `init-statement` to be `int i = 0`
- `condition` to be `true`
- `statements` to be `int i = 1; break;`
- `expression` to be empty

then you get a `for` loop that produces a build error and a `while` loop that builds and executes with no problems.

```
int main() {
    for (int i = 0; true; ) {
        int i = 1; break;     // build error
    }
    return 0;
}
```

```
int main() {
    {
        int i = 0;

        while (true) {
            int i = 1; break;
        }
    }
    return 0;
}
```

Because a `for` loop pretty much just abbreviates a `while` loop, any code that uses a `for` loop can be rewritten to use a `while` loop and most sensible code that is written with a `while` loop can be rewritten with a `for` loop. However, it often feels more natural to use a `for` loop in some situations and more natural to use a `while` loop in other situations.

- Suppose that we are writing code that stores a non-negative integer value in a variable `N`, and we want some lines of code to execute `N` times. This seems like the perfect time to use a `for` loop.

```
        for (int i = 0; i < N; ++i) {
            // lines of code to execute N times
        }
```

- Suppose that we want some lines of code to repeat, but it is difficult or impossible to know how many times they will need to execute. This seems like a good moment to use a `while` loop.
  - Waiting for a user to respond correctly to a prompt gives an example where it is impossible to know how many times the code will need to execute. See example 8.3.6.
  - Printing the first `N` primes is an example where it is difficult to know how many natural numbers we will have to check for primality. See example 8.6.6.

Example 8.5.4. Although the objective of the following code is unclear, I still object to it using a `for` loop. This is because `for (int j = 0; j < 100; ++j)` strongly suggests that what is within the `for` loop's braces will execute 100 times. Unfortunately, this is not true, so the code is misleading. The second version of the code is less misleading.

```
#include <iostream>

int main() {
    for (int j = 0; j < 100; ++j) {
        std::cout << j << std::endl;

        if (j < 32) {
            j *= 2;
        }
        else {
            j += 3;
        }
    }
    return 0;
}
```

```
#include <iostream>

int main() {
    int j = 0;

    while (j < 100) {
        std::cout << j << std::endl;

        if (j < 32) {
            j  =  2 * j  +  1;
        }
        else {
            j += 4;
        }
    }
    return 0;
}
```

Example 8.5.5. To help become familiar with `for` loops, try predicting the output of the following code.

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

int main() {
    for (int j = 0; j < 10; ++j) {
        cout << j << '␣';
    }
    cout << endl;

    for (int j = 1; j <= 10; ++j) {
        cout << j << '␣';
    }
    cout << endl;

    for (int j = 1; j < 10; ++j) {
        cout << j << '␣';
    }
    cout << endl;

    for (int j = 0; j <= 10; ++j) {
        cout << j << '␣';
    }
    cout << endl;

    for (int j = 0; j < 10; j += 2) {
        cout << j << '␣';
    }
    cout << endl;

    for (int j = 10; j > 0; --j) {
        cout << j << '␣';
    }
    cout << endl;

    return 0;
}
```

The first of these `for` loops is the most common. In math, one often counts from `1` to `N` inclusively, but in computer science, most frequently, one counts from `0` up to and not including `N`. For example, this is frequently how we will loop through a `std::vector` (section 11), and this is how we can loop through a `std::string`. In fact, when looping through these containers, it can be useful to initialize two variables...

Example 8.5.6. In the following `for` loop, look carefully at what is written before the first semicolon: `size_t i = 0, N = s.length()`. This code declares and initializes two `size_t`s. `i` is a `size_t` because it is used to index the `string`. `N` is a `size_t` because it stores the size/length of the `string`.

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

int main() {
    string s = "abcdefgh";

    for (size_t i = 0, N = s.length(); i < N; ++i) {
        s[i] += 'A' - 'a';
    }
    cout << s << endl;

    return 0;
}
```

Example 8.5.7. To practice using `for` loops, try to write a program that declares and defines a `const int n`, initializes it to a value bigger than or equal to `3`, and then draws a triangle using that many lines. For example, when `n` is initialized with `3`, you want the output to be...

```
  *
 * *
*****
```

When `n` is initialized with `4`, you want the output to be...

```
   *
  * *
 *   *
*******
```

When `n` is initialized with `5`, you want the output to be...

```
    *
   * *
  *   *
 *     *
*********
```

When `n` is initialized with `6`, you want the output to be...

```
     *
    * *
   *   *
  *     *
 *       *
***********
```

Example 8.5.8. If one wants to produce some equally spaced `double`s, it is normally best to avoid using a `for` loop directly with `double`s. The output of the code below is as follows.

```
0 0.125 0.25 0.375 0.5 0.625 0.75 0.875 after looping... 1
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 after looping... 1.1
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
```

In the first `for` loop, `epsilon` stores 0.125 perfectly and the `for` loop prints `0` up to and not including `1` equally spaced by `0.125`. In the second `for` loop, `epsilon` cannot store 0.1 perfectly and the `for` loop appears to print `0` up to and including `1`. The last number printed is actually `1 - pow(2, -53)`. Therefore, if one wants to use the numbers `0` up to and not including `1` and have them as close to equally spaced by 0.1 as possible, it is best to write a loop involving `int`s and divide them by `10.0`. If you type `#include <iomanip>` and `cout << setprecision(17)`, you will find even more evidence that this does a better job.

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

int main() {
    double d, epsilon;

    d = 0.0;
    epsilon = 0.125;

    // 0.125 is stored perfectly...
    for (; d < 1; d += epsilon) {
        cout << d << ' ';
    }
    cout << "after looping... " << d << endl;

    d = 0.0;
    epsilon = 0.1;

    // 0.1 is not stored perfectly...
    for (; d < 1; d += epsilon) {
        cout << d << ' ';
    }
    cout << "after looping... " << d << endl;

    // This is better...
    for (int i = 0; i < 10; ++i) {
        cout << i / 10.0 << ' ';
    }
    cout << endl;

    return 0;
}
```

Example 8.5.9. The following code, which prints `true`, is inspired by equation (5.7.6.2) of example 5.7.6.1.

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

int main() {
    double d = 0.0;

    for (int e = 0; e <= 52; e += 2) {
        d += pow(2.0, -e);
    }
    cout << boolalpha << (d == 4.0 / 3.0) << endl;

    return 0;
}
```

8.6 Further Examples

Prime numbers are sufficiently complicated that they provide some great examples of using control flow. We will solve the following problems in this section.

- Printing the primes numbers which are less than or equal to a specified natural number.
- Determining whether a specific natural number is prime or not.
- Printing the first N primes.

Note that the first and last problem are different. The prime numbers less than or equal to 19 are

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,$$

but the first 19 primes numbers are...

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 41,\ 43,\ 47,\ 53,\ 59,\ 61,\ 67.$$

The latter problem is more difficult because it is not so clear how far into the natural numbers one has to look before enough primes will be found (unless you know some advanced number theory). Would you have guessed that looking until 67 is enough to find 19 primes? This is a good example where many people would choose a `while` loop over a `for` loop.

To get going we need to know what a prime number is.

Definition 8.6.1. A natural number, that is, a number in the set $\{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ \dots\}$, is said to be *prime* if and only if...

- it is not equal to 0 or 1;
- the only natural numbers that divide it exactly are 1 and itself.

Alternatively, if you understand the Sieve of Eratosthenes, then you can say that the prime numbers are those numbers which are not crossed out by this algorithm. This is a good working definition because it also provides one of the fastest algorithms for computing primes below a specified number.

Example 8.6.2. To run the Sieve of Eratosthenes we need a way to store something equivalent to the grid of numbers. With our current tools, it is simplest to make a `std::string` called `is_prime` consisting of the characters `'t'` and `'f'`, standing for "true" and "false", respectively. A `std::string` storing the chracters `"ffttftftfffftft"` would be storing the fact that 0 is not prime (`f`), 1 is not prime (`f`), 2 is prime (`t`), 3 is prime (`t`), 4 is not prime (`f`), 5 is prime (`t`), 6 is not prime (`f`), 7 is prime (`t`), 8 is not prime (`f`), 9 is not prime (`f`), 10 is not prime (`f`), 11 is prime (`t`), 12 is not prime (`f`), and 13 is prime (`t`). The sieve begins as though every numbers is prime. We have made the `std::string` `N + 1` characters long so that the valid indices are `0`, `1`, `2`, ..., `N`. We immediately mark `0` and `1` as not prime. Then we use nested `for` loops to mark off products as not prime. Then we print out the values that are still marked as prime.

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

int main() {
    const unsigned int N = 19;

    // Prints the primes numbers which are less than or equal to N
    // using https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

    string is_prime(N + 1, 't');

    is_prime[0] = 'f';
    is_prime[1] = 'f';

    for (size_t i = 2; i <= N; ++i) {
        for (size_t j = 2; i * j <= N; ++j) {
            is_prime[i * j] = 'f';
        }
    }
    for (size_t p = 0; p <= N; ++p) {
        if (is_prime[p] == 't') {
            cout << p << '␣';
        }
    }

    cout << endl;
    return 0;
}
```

One can arrive at the nested `for` loops by thinking about many consecutive `for` loops like below. Notice that the condition used in the inner `for` loop — `i * j <= N` — very clearly ensures valid indices are used for `is_prime`. Because the numbers `i`, `j`, and `p` are indices, we use `size_t`s.

```cpp
    // "Cross out" multiples of 2 except 2.
    // "except 2" is the reason j starts at 2 and not 1.
    for (size_t j = 2; 2 * j <= N; ++j) { is_prime[2 * j] = 'f'; }

    // "Cross out" multiples of 3 except 3.
    // "except 3" is the reason j starts at 2 and not 1.
    for (size_t j = 2; 3 * j <= N; ++j) { is_prime[3 * j] = 'f'; }

    // "Cross out" multiples of 4 except 4.
    for (size_t j = 2; 4 * j <= N; ++j) { is_prime[4 * j] = 'f'; }

    // "Cross out" multiples of 5 except 5.
    for (size_t j = 2; 5 * j <= N; ++j) { is_prime[5 * j] = 'f'; }

    // Keep going this way...
```

Example 8.6.3. The previous code can be refined a little. This version only "crosses out" multiples of numbers already confirmed to be prime. It also crosses out multiples of p starting at p^2 instead of $2p$. Finally, it also prints the primes as it computes them.

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

int main() {
    const unsigned int N = 19;

    // Prints the primes numbers which are less than or equal to N
    // using https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

    string is_prime(N + 1, 't');

    for (size_t p = 2; p <= N; ++p) {
        if (is_prime[p] == 't') {
            cout << p << ' ';

            for (size_t j = p; j * p <= N; ++j) {
                is_prime[j * p] = 'f';
            }
        }
    }

    cout << endl;
    return 0;
}
```

Although the Sieve of Eratosthenes is fast, without using some number theory to know how far to look, it does not lend itself very well to finding the first `N` primes. There are ways to circumvent this problem, but we will take a different approach which showcases some nice control flow.

First, we will solve the previous problem in a new way. Our new strategy will be to go through the numbers 1 to N and for each number to ask, "is it prime?" This creates a sub-problem: if n is a natural number, can we determine if it is prime? That is, can we check whether 1 and n are the only natural numbers dividing it? The next example solves this subproblem.

Example 8.6.4. In this example we write code that answers the question "is n prime?" We start off by labelling the number as prime. Then we loop through the potential non-trivial divisors d, the numbers 2 to $n-1$, checking whether any of them divide n. If one of them divides n, then we mark n as not prime and `break` from the `for` loop.

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

int main() {
    const unsigned int n = 35; // Needs to be 2 or bigger.

    // Determines whether n is prime by checking
    // for divisors between 2 and n - 1.

    bool n_is_prime = true;

    for (unsigned int d = 2; d < n; ++d) {
        if (n % d == 0) {
            cout << n << " == " << d << " * " << n / d;

            n_is_prime = false;
            break;
        }
    }
    if (n_is_prime) {
        cout << n << " is prime" << endl;
    }
    else {
        cout << " so " << n << " is not prime" << endl;
    }
    return 0;
}
```

Two remarks are necessary.

- This code will incorrectly say that 0 and 1 are prime.

  It should not be used to answer the question for those two numbers.

- This code can be made quicker by looking for divisors from 2 to $\sqrt{n}$ instead of 2 to $n-1$.

We can now solve the first problem in a different way by wrapping up the previous example in a `for` loop.

Example 8.6.5. This code is obtained by wrapping the previous example up in a `for` loop (and removing some print statements).

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

int main() {
    const unsigned int N = 19;

    // Prints the primes numbers which are less than or equal to N
    // by checking whether each number less than or equal to N
    // has a non-trivial divisor.

    for (unsigned int n = 2; n <= N; ++n) {
        bool n_is_prime = true;

        for (unsigned int d = 2; d < n; ++d) {
            if (n % d == 0) {
                n_is_prime = false;
                break;
            }
        }
        if (n_is_prime) {
            cout << n << ' ';
        }
    }
    cout << endl;
    return 0;
}
```

Example 8.6.6. We are finally ready to give a solution to "printing the first `N` primes". It makes sense to record how many primes we have found as we progress. For this, we use a variable called `primes_so_far`. We then use very similar code to before over and over again as long as `primes_so_far` is less than `N`. For readability reasons, we have relabelled `n` as `candidate`. The code is auditioning each number one by one to see whether it passes the test to be prime.

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

int main() {
    const unsigned int N = 19;

    // Prints the first N primes by
    // checking whether each number
    // has a non-trivial divisor
    // until N primes have been found.

    unsigned int primes_so_far = 0;
    unsigned int candidate = 2;

    while (primes_so_far < N) {
        bool candidate_is_prime = true;

        for (unsigned int d = 2; d < candidate; ++d) {
            if (candidate % d == 0) {
                candidate_is_prime = false;
                break;
            }
        }
        if (candidate_is_prime) {
            cout << candidate << '␣';
            ++primes_so_far;
        }
        ++candidate;
    }
    cout << endl;
    return 0;
}
```

It feels necessary to point out that this is far from the fastest code that one can write. Using some number theory together with the sieve would be faster. Even this code can be optimized to be much faster. One could use the square root as mentioned in example 8.6.4, and if one uses a `std::vector` to store the primes found so far, one can make sure to only loop through these as potential divisors.

8.7 Review questions

1. (a) The following code intends to help a user calculate the absolute value of `int`s.
Say if the following code encounters a build error, a runtime error, undefined behavior, or any other issues.

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

int main() {
    cout << "Give␣me␣an␣int␣and␣I␣will␣";
    cout << "tell␣you␣its␣absolute␣value:␣";

    int i; cin >> i;

    cout << "The␣absolute␣value␣of␣" << i << "␣is␣";

    if (i > 0) {
        cout <<  i << endl;
    }
    if (i < 0) {
        cout << -i << endl;
    }

    return 0;
}
```

(b) Say if the following code encounters a build error, a runtime error, or undefined behavior. If not, write down the output.

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

int main() {
    int year = 2000;

    if ((year % 400 == 0 || year % 4 == 0) && year % 100 != 0) {
        cout << year << "␣is␣a␣leap␣year." << endl;
    }
    else {
        cout << year << "␣is␣not␣a␣leap␣year." << endl;
    }

    return 0;
}
```

(c) Say if the following code encounters a build error, a runtime error, or undefined behavior. If not, write down the output.

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

int main() {
    int i = 28;

    if (i = 8) {
        cout << 1 << endl;
    }
    else {
        cout << 2 << endl;
    }

    return 0;
}
```

(d) Say if the following code encounters a build error, a runtime error, or undefined behavior. If not, write down the output.

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

int main() {
    string s = "AAARGH!!!";

    if (s.find("???")) { cout << 1 << endl; }
    if (s.find("RGH")) { cout << 2 << endl; }
    if (s.find("AAA")) { cout << 3 << endl; }

    return 0;
}
```

2. (a) Say if the following code encounters a build error, a runtime error, or undefined behavior. If not, write down the output.

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

int main() {
    int a = 0;
    int b = 1;

    {
        int tmp = a;
        a = b;
        b = tmp;
    }

    cout << tmp << '␣' << a << '␣' << b << endl;

    return 0;
}
```

(b) Say if the following code encounters a build error, a runtime error, or undefined behavior. If not, write down the output.

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

int main() {
    int a = 0;
    int b = 1;

    {
        int tmp = a;
        a = b;
        b = tmp;
    }

    cout << a << '␣' << b << endl;

    return 0;
}
```

(c) Say if the following code encounters a build error, a runtime error, or undefined behavior. If not, write down the output.

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

int main() {
    int i = 0;
    int j = 1;
    int k = 2;

    if (j < k) {
        int i = j;
        j = k;
        k = i;
    }

    cout << i << '␣' << j << '␣' << k << endl;

    return 0;
}
```

3. (a) What is the output of the following code?

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

int main() {
    string s = "abcd";

    for (size_t i = 0, N = s.length(); i < N; ++i) {
        s.pop_back();
    }

    cout << s.length() << endl;
    return 0;
}
```

(b) What is the output of the following code?

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

int main() {
    string s = "abcd";

    for (size_t i = 0; i < s.length(); ++i) {
        s.pop_back();
    }

    cout << s << endl;
    return 0;
}
```

(c) What is the output of the following code?

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

int main() {
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            cout << i << ",␣" << j << endl;
        }
    }
    return 0;
}
```

(d) What is the output of the following code?

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

int main() {
    for (int j = 0; j < 31; ++j) {
        cout << j << '␣';

        if (j < 8) {
            j *= 2;
        }
        else {
            j += 3;
        }
    }
    cout << endl;
    return 0;
}
```

9 Functions

The act of coding amounts to defining and calling functions and so functions are very important! A strong understanding of functions and function calls leads to learning coding languages more easily and efficiently, so this is the part of the course most deserving of your attention.

9.1 An example of almost everything

Save for function comments, the next code snippet provides a talking point for pretty much everything that we need to learn. It will be referred to as "the uber-example".

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

int my_max(int i, int j);
void print(double d);

int main() {
    int a, b, c, i, j, k;

    a = 0; b = 1;
    c = my_max(a, b);
    cout << c << endl;

    a = 3; b = 2;
    c = my_max(a, b);
    cout << c << endl;

    i = 5; j = 4;
    k = my_max(j, i);
    cout << k << endl;

    print(1.01);
    return 0;
}

int my_max(int i, int j) {
    cout << "my_max␣has␣been␣called.␣";

    if (i > j) {
        return i;
    }
    return j;
}

void print(double d) {
    cout << "printing...␣";
    double tmp = d; cout << tmp << endl;
    // return;
}
```

This code builds on all compilers and executes without encountering errors. The output produced is as follows.

```
1   my_max has been called. 1
2   my_max has been called. 3
3   my_max has been called. 5
4   printing... 1.01
```

First, I will introduce the relevant language without too much explanation of the concepts involved. Then I will get into the nitty-gritty of how everything in the uber-example works.

Example 9.1.1. Lines 4 and 5 of the uber-example provide examples of *declaring* functions.

```
4   int my_max(int i, int j);
5   void print(double d);
```

Example 9.1.2. Lines 26 to 33 and lines 35 to 39 of the uber-example provide examples of *defining* functions.

```
26  int my_max(int i, int j) {
27      cout << "my_max␣has␣been␣called.␣";
28
29      if (i > j) {
30          return i;
31      }
32      return j;
33  }
34
35  void print(double d) {
36      cout << "printing...␣";
37      double tmp = d; cout << tmp << endl;
38      // return;
39  }
```

Remark 9.1.3. These examples fulfill a promise made in remark 5.2.1.2, that you would see it is possible to separate a function declaration from its definition. On the other hand, it is also possible to declare and define a function at the same time.

Example 9.1.4. Lines 7 to 24 of the uber-example provide an example of declaring *and* defining a function simultaneously. The point is that a definition of a function serves as a declaration if there is not a declaration elsewhere. This works for all functions, not just `main`.

```
7   int main() {
8       int a, b, c, i, j, k;
9
10      a = 0; b = 1;
11      c = my_max(a, b);
12      cout << c << endl;
13
14      a = 3; b = 2;
15      c = my_max(a, b);
16      cout << c << endl;
17
18      i = 5; j = 4;
19      k = my_max(j, i);
20      cout << k << endl;
21
22      print(1.01);
23      return 0;
24  }
```

Example 9.1.5. Lines 11, 15, 19, and 22 of the uber-example provide examples of *function calls*. *Calling* a function is just the professional way to talk about using a function.

```
11      c = my_max(a, b);
15      c = my_max(a, b);
19      k = my_max(j, i);
22      print(1.01);
```

Build Error 9.1.6. *It is a build error to call a function before its declaration.*

Example 9.1.7. Commenting out line 4 or line 5 will create a build error for the uber-example.

Build Error 9.1.8. *It is a build error to call a function when it is not defined. More specifically, doing so creates a linking error.*

Example 9.1.9. Commenting out lines 26 to 33 or lines 35 to 39 will create a build error for the uber-example.

Remark 9.1.10. It is legal to declare a function multiple times (and this legality is useful).

Build Error 9.1.11. *It is a build error to define a (non-inline) function more than once.*

Example 9.1.12. Line 26 and line 35 of the uber-example provide examples of function *parameters*: `i`, `j`, and `d`. Function parameters exist to be used in function definitions.

```
26  int my_max(int i, int j) {
35  void print(double d) {
```

These parameters are also listed in the function declarations, but one could replace lines 4 and 5 by the following without creating a build error: that is, the declarations do not need the names of the parameters. In fact, even the definitions only need the names when they are used in the remainder of the definition (but this is almost always the case in useful examples).

```
4   int my_max(int, int);
5   void print(double);
```

Example 9.1.13. Lines 11, 15, 19, and line 22 of the uber-example provide examples of function *arguments*: `a`, `b`, `a` (again), `b` (again), `j`, `i`, and `1.01`. Unlike function parameters, which are used in function definitions, function arguments are provided to a function at the moment the function is called, that is, when it is used.

```
11      c = my_max(a, b);
15      c = my_max(a, b);
19      k = my_max(j, i);
22      print(1.01);
```

Remark 9.1.14. `i` and `j` appear in the uber-example as both function parameters and function arguments. This is intentional and is to clarify that everything still makes sense even when such name conflicts exist. However, you need to wait until the nitty-gritty details to resolve any confusion this has caused.

Example 9.1.15. Function declarations like on line 4 and 5 of the uber-example provide what are called *function signatures*. Function signatures show up in documentation as well as written code.

```
4   int my_max(int i, int j);
5   void print(double d);
```

A function signature consists of:

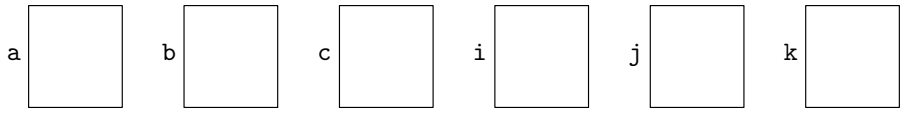
- The function name. For example, `my_max` and `print`.
- The number and type of the parameters. For example, two `int`s and one `double`.
- The return type. For example, `int` and `void`.

Definition 9.1.16. `void` is a fundamental type with an empty set of values. You cannot declare a variable of type `void`. Therefore, a return type of `void` indicates that no value is returned.

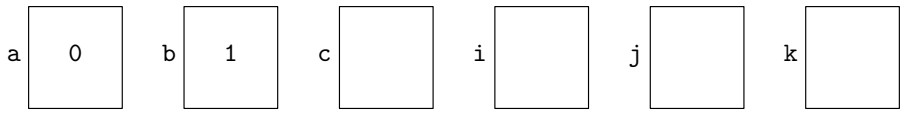
Now we have all the most essential terminology in place, we need to understand what is going on in the uber-example. Most importantly, we need to understand how the funtion calls are executed. We will execute the code line by line, but as we will see, this will involve a little jumping around. For convenience, here is the uber-example again in full.

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

int my_max(int i, int j);
void print(double d);

int main() {
    int a, b, c, i, j, k;

    a = 0; b = 1;
    c = my_max(a, b);
    cout << c << endl;

    a = 3; b = 2;
    c = my_max(a, b);
    cout << c << endl;

    i = 5; j = 4;
    k = my_max(j, i);
    cout << k << endl;

    print(1.01);
    return 0;
}

int my_max(int i, int j) {
    cout << "my_max␣has␣been␣called.␣";

    if (i > j) {
        return i;
    }
    return j;
}

void print(double d) {
    cout << "printing...␣";
    double tmp = d; cout << tmp << endl;
    // return;
}
```

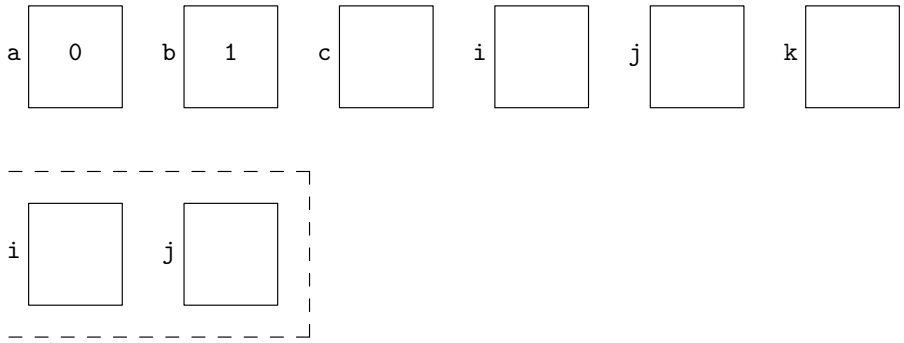
Code execution starts on line 8. Six variables are declared and defined.



Line 10 initializes `a` and `b` with values of `0` and `1`, respectively.



Line 11 is our first function call. We learn our first thing about function calls: **when a function is called a function scope is introduced to contain the parameters and any other variables that are declared in the function definition.** Here is the picture.



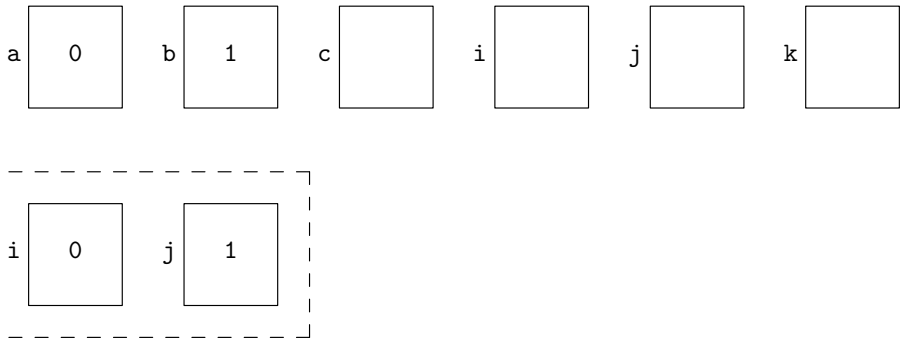
Just like before, we have used a dashed box to indicate a scope.

Next, **arguments are assigned to parameters.** The parameters of `my_max` are `i` and `j`. The corresponding arguments used in the function call are `a` and `b`.

These first two steps behave like the following lines of code have executed.

```
1  int i = a;
2  int j = b;
```

The picture is now as follows.



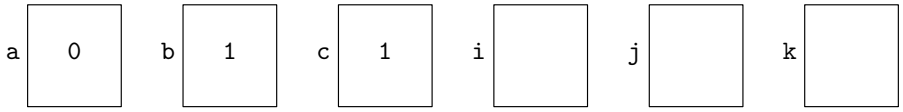
Now that `i` and `j` have been initialized, the **body of the function**, what is within the braces of the function definition, can execute. Line 27 causes the first output to the console:

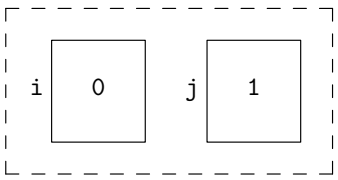
```
1    my_max has been called.
```

Line 29 compares the values of the parameters `i` and `j`. Because `i > j` evaluates to `false`, line 30 does not execute. We are left with line 32. There are two things to say.

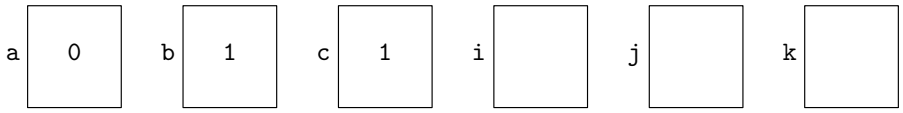
- If an expression is returned and the function call is assigned to a variable, the code execution behaves as though **the expression is assigned to that variable.**
- **A `return` statement always ends a function call,** even if it is not the last line of code in the function body.

To further explain the first bullet point, in our current situation `j` is returned and line 12 makes the assignment `c = my_max(a, b)`, and so it is though the assignment `c = j` is performed. This updates `c`'s value to `1`.





Because of the second point, the function call ends, the function scope is destroyed, and the variables in our code are left as below.



The function call of line 11 behaved a lot like the following code.

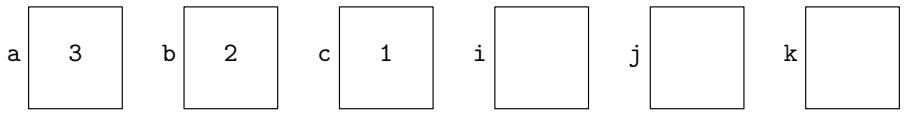
```
11      {
12          int i = a; int j = b;
13          cout << "my_max␣has␣been␣called.␣";
14
15          if (i > j) {
16              c = i;
17          }
18          c = j;
19      }
```

This is a good analogy, but it is not perfect. Function scope is stricter (not nested) than the scope introduced by braces, and braces do not handle the naming conflict we encounter on line 18 in the way that function scopes do. Moreover, as we are about to see, the `return` statement needs to be handled a little more carefully.

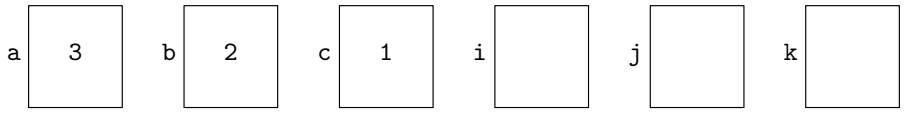
We processed line 11. Line 12 prints out `c`'s value. At this point we have the following output.

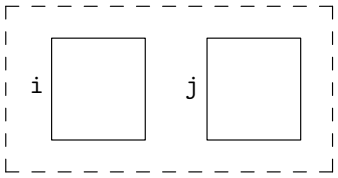
```
1    my_max has been called. 1
```

Line 14 assigns the values `3` and `2` to `a` and `b`, respectively.



We can now execute line 15 similarly line 11...
Introduce function scope storing the parameters.

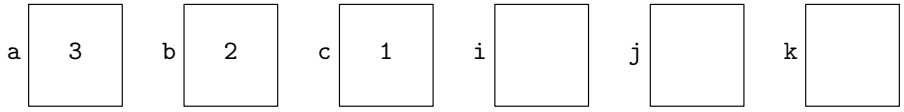


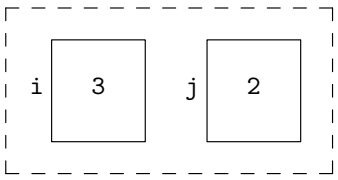


Assign arguments to parameters.
These first two steps behave like the following lines of code have executed.

```
1    int i = a;
2    int j = b;
```

The picture is now as follows.



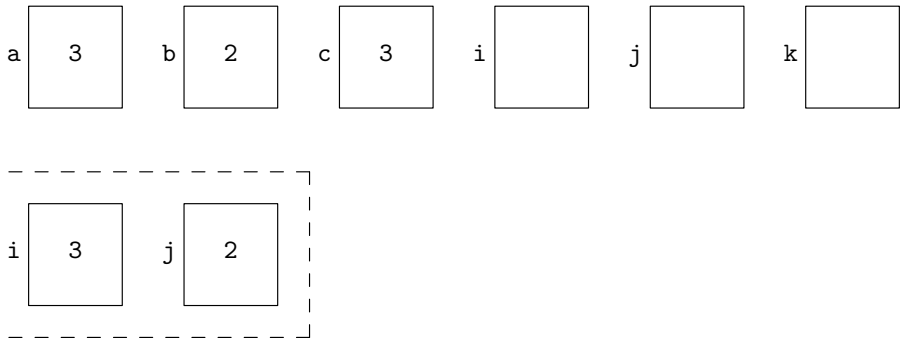


Execute the function body.
Line 27 executes to produce more output so that the console looks as follows.

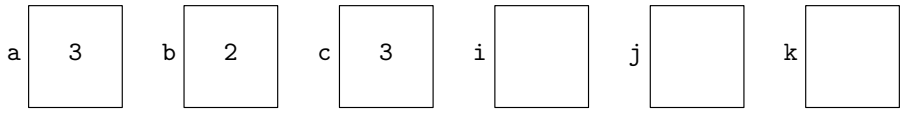
```
1    my_max has been called. 1
2    my_max has been called.
```

`i > j` evaluates to `true` and so we encounter `return i` on line 30.

This first behaves like an **assignment** `c = i` so that `c`'s value is updated to `3`.



Then **the `return` statement ends the function call.** This destroys the function scope so we are left with the following picture.



The function call of line 15 did **not** behave like the following code because `c = j` does not execute.

```
15      {
16          int i = a; int j = b;
17          cout << "my_max␣has␣been␣called.␣";
18
19          if (i > j) {
20              c = i;
21          }
22          c = j;
23      }
```

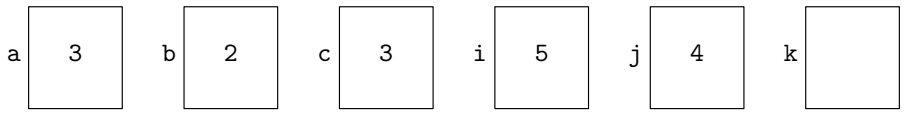
If you want to improve the analogy, you need something like the following code. Even though `goto` statements are actively discouraged, in this analogy they are highlighting that **a `return` statement always ends a function call.**

```
15      {
16          int i = a; int j = b;
17          cout << "my_max␣has␣been␣called.␣";
18
19          if (i > j) {
20              c = i; goto END_OF_BRACES;
21          }
22          c = j; goto END_OF_BRACES;
23      }
24      END_OF_BRACES:
```

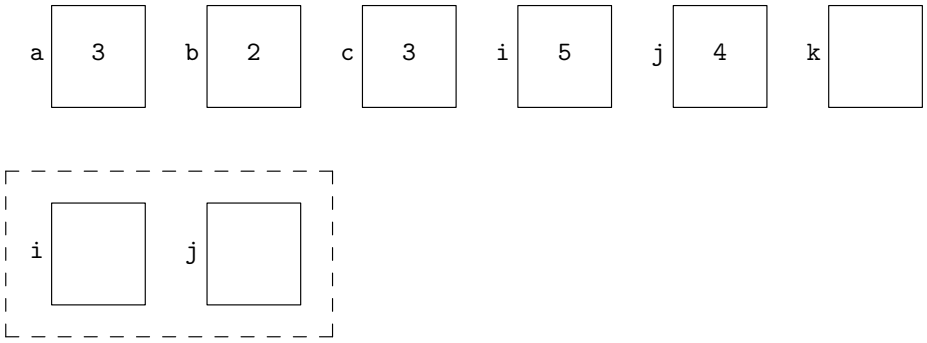
We processed line 15. Line 16 prints out `c`'s value. At this point we have the following output.

```
my_max has been called. 1
my_max has been called. 3
```

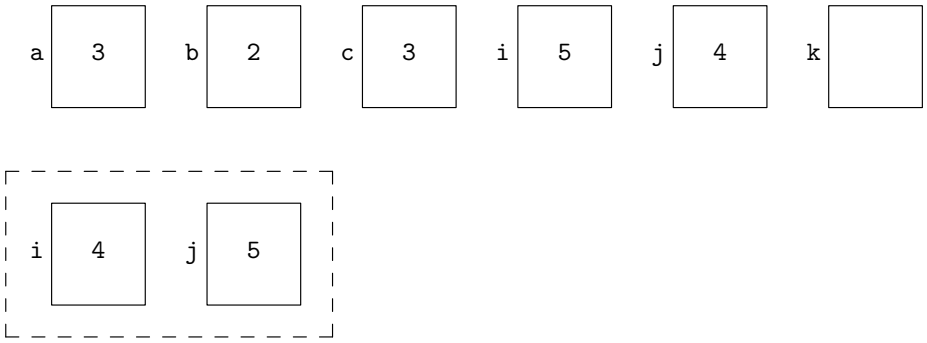
Line 18 assigns the values `5` and `4` to `i` and `j`, respectively.



We can now execute line 19 similarly line 11 and line 15...
Introduce function scope storing the parameters.



Assign arguments to parameters.
The picture is now as follows.



Something clever has happened here. The following lines, on their own, are highly confusing.

```
int i = j;
int j = i;
```

Moreover, if they were within a scope introduced by braces, with variables outside the scope called `i` and `j`, then the first initialization would use the `j` outside the scope, and the second initialization would use the `i` *inside the scope*. **This is not at all what happens in the function call.** Here is how to think about it.

- The function definition says...

```
26    int my_max(int i, int j) {
```

- The function call says...

```
19        k = my_max(j, i);
```

- Look at the first argument (`j`) and first parameter (`i`).
  This says to use `main`'s `j` to initialize `my_max`'s `i`.

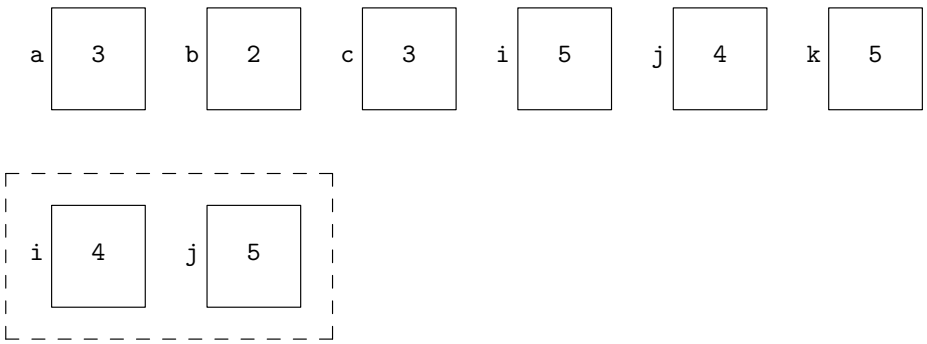
- Look at the second argument (`i`) and second parameter (`j`).
  This says to use `main`'s `i` to initialize `my_max`'s `j`.

After overcoming this hurdle, the rest of the execution is like before.

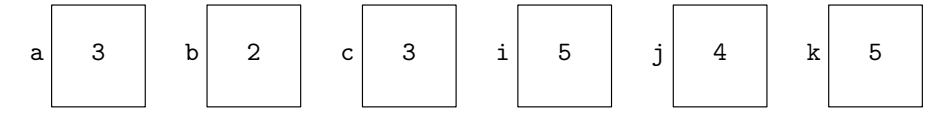
Execute the function body.
Line 27 executes to produce more output so that the console looks as follows.

```
1    my_max has been called. 1
2    my_max has been called. 3
3    my_max has been called.
```

`i > j` evaluates to `false` and so we do not execute line 30 and we encounter `return j` on line 32. This first behaves like an **assignment** `k = j` (using the `j` in the function scope) so that `k`'s value is updated to `5`.



Then **the `return` statement ends the function call.** This destroys the function scope so we are left with the following picture.



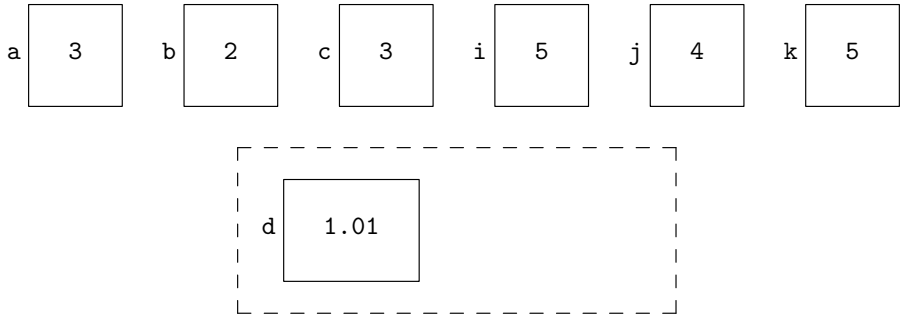
We processed line 19. Line 20 prints out `k`'s value. At this point we have the following output.

```
1  my_max has been called. 1
2  my_max has been called. 3
3  my_max has been called. 5
```

Line 22 executes a lot like

```
22      {
23          double d = 1.01;
24          cout << "printing...␣";
25          double tmp = d; cout << tmp << endl;
26      }
```

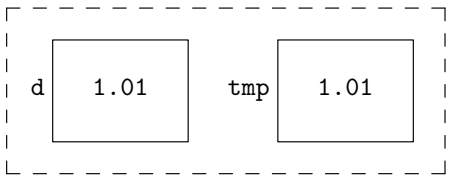
Here is the picture after **introducing the function scope to store the parameter `d` and assigning the argument `1.01` to the parameter `d`.**



While **executing the function body** both lines 36 and 37 produce output that leaves the console saying...

```
1  my_max has been called. 1
2  my_max has been called. 3
3  my_max has been called. 5
4  printing... 1.01
```

...but the purpose of this example is to point out the variable `tmp` used within the function scope. Any variables declared in the function body will pop into existence within the function scope when the function is called and will disappear once the function call is over.



The final reason for this example is to call attention to the fact that `return` can still be used in a

function whose return type is `void`. You cannot `return` an expression, but the `return` keyword can still be useful for ending the function call before the end of the function body. If it is not typed, it is as though it is written at the very end of the function body. Therefore uncommenting line 38 changes nothing.

Build Error 9.1.17. *Because function scope is stricter than the scope introduced by braces, it is a build error for a function body to use a variable which is not a parameter and which has not been declared within the function body (with the exception of global variables which are discouraged in most scenarios).*

Example 9.1.18. The function `f` has no parameters and its definition declares no variables, so if `f` was called, its function scope would be empty. Its function scope would not be nested within `main`'s scope, and so it would not be able to access `main`'s `i`. Line 9 causes a build error.

```
1   void f();
2
3   int main() {
4       int i = 0;
5       return 0;
6   }
7
8   void f() {
9       i = 1;
10  }
```

Undefined Behavior 9.1.19. *Suppose `f` is a function with a non-`void` return type and that `v` is a variable with type equal to the return type of `f`. If calling `f(args...)` leads to reaching the closing brace `}` of `f`'s definition without encountering a `return` statement, then `v = f(args...)` produces undefined behavior.*

Example 9.1.20. The following code encounters undefined behavior because the careless control flow in the bad definition of `abs_val` does not say what is returned when `i == 0`.

```
1   int abs_val(int i) {
2       if (i > 0) {
3           return i;
4       }
5       if (i < 0) {
6           return -i;
7       }
8   }
9
10  int main() {
11      int v = abs_val(0);
12      return 0;
13  }
```

9.2 Function comments

There is a very common way to comment functions.

Function comments are often written inside `/** */`.

- They start with a useful description of the function and what it does.
- For each parameter, one should write `@param`, the parameter name, and provide a description of its relevance.
- When the function's return type is not `void`, one should write `@return` and provide a description of what is returned.

The length of the descriptions depends on the complexity of the function. They should be detailed enough so that most people who know how to read function comments can deduce what the function does without having to look at and/or understand its definition.

Example 9.2.1. Since the `my_max` function is fairly simple, the descriptions in the function comment are short.

```
1  #include <iostream>
2  using namespace std;
3
4  /**
5   This function calculates
6   the maximum of two ints.
7   @param i : the first int
8   @param j : the second int
9   @return the maximum of i and j
10 */
11 int my_max(int i, int j);
12
13 int main() {
14     cout << my_max(1, 2) << endl;
15     return 0;
16 }
17
18 int my_max(int i, int j) {
19     if (i > j) {
20         return i;
21     }
22     return j;
23 }
```

It seems like XCode understands these function comments well and will provide warnings when they are incompatible with the function declaration but that VS 2022 is less useful in this respect.

9.3 Examples

Many examples of functions with function comments can be found here.

9.4 Pure functions and procedures

Definition 9.4.1.

- Functions which `return` a value (i.e. their return type is not `void`) but do not have side effects (like printing to the console or changing the value of variables even when no assignment is made using the return value) are called *pure functions*.
- Functions which do not `return` a value (i.e. their return type is `void`) but which have side effects (like printing to the console or changing the value of variables) are called *procedures*.
- When writing functions, it is normally most clear when the function is either a procedure or a pure function. Some functions are neither and such functions can be useful, but they might be confusing for a user.

Example 9.4.2. In the following code, `abs_val` is a pure function. It returns an `int`. Moreover, it does not print anything to the console and it does not change any variables' values in `main` when an assignment is not used.

```
1  #include <iostream>
2  using namespace std;
3
4  /**
5   This pure function returns
6   the absolute value of an int.
7   @param i : the int
8   @return the absolute value of i
9  */
10 int abs_val(int i) {
11     if (i > 0) {
12         return i;
13     }
14     return -i;
15 }
16
17 int main() {
18     int i = -28;
19
20     cout << "The␣absolute␣value␣of␣" << i << "␣is␣";
21     cout << abs_val(i) << endl;
22
23     return 0;
24 }
```

Example 9.4.3. In the following code, `print` is a procedure: its return type is `void` and it prints to the console. The standard-defined `std::swap` is also a procedure. Looking it up on the internet, one finds its return type is `void`. Moreover, it changes the values of `i` and `j` in `main` even though no assignment is used (since its return type is `void` using an assignment would give a build error).

```
#include <iostream>
#include <utility>

/**
 This procedure prints
 an int to the console
 followed by '\n'.
 @param i : the int to print
*/
void print(int i) {
    std::cout << i << '\n';
}

int main() {
    int i = 0;
    int j = 1;

    print(i); print(j);

    std::swap(i, j);

    print(i); print(j);

    return 0;
}
```

Example 9.4.4. The following function `my_getline_ish` is neither a pure function or a procedure.

- It has a side effect — changing the input buffer — so it is not a pure function.
- It returns a `std::string` so it is not a procedure.

The code is on the next page.

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

/**
 This function extracts characters from the
 input buffer up to and not including '\n',
 and returns them in a string
 after deleting the following '\n'.
 @return a string containing the characters
 extracted from the input buffer
*/
string my_getline_ish() {
    string s;

    while (cin.peek() != '\n') {
        s += cin.get();
    }
    cin.ignore();

    return s;
}

int main() {
    cout << "I'll␣repeat␣what␣you␣say..." << endl;
    cout << my_getline_ish() << endl;

    return 0;
}
```

9.5 Remember that `main` is a function

Only for the adventurous, here is a fairly bizarre example. Recall that `main` is a function. One can perform recursion with `main`! This program ends with an exit code of `4`.

```cpp
int i = 11;

int main() {
    if (i % 2 == 0) {
        return i;
    }
    else {
        i /= 2;
        return main() + 1;
    }
}
```

9.6 Reviewing syntax

For *declaring* functions, the syntax is...

```
return_type function_name(param1_type, param2_type, ...);
```

or

```
return_type function_name(param1_type param1_name, param2_type param2_name, ...);
```

For *defining* functions, the syntax is...

```
return_type function_name(param1_type param1_name, param2_type param2_name, ...) {
    instructions
}
```

For *calling* a function the syntax is...

```
function_name(argument1, argument2, ...);
```

9.7 Reviewing function calls

- Introduce function scope.
- Assign arguments to parameters.
- Execute the instructions of the function body.
- If the function's return type is not `void` and the function call is assigned to a variable, make sure to update the assigned-to variable using whatever expression is returned.
- End the function call and destroy the function scope.
- **Never claim that two values are returned in a single function call!**

9.8 Review questions

1. Consider the following code.

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

/**
 This function squares a double.
 @param x : the double to square
 @return x^2
*/
double square(double x) {
    return x * x;
}

int main() {
    double d = 3.0;
    cout << d << "␣squared␣is␣" << square(d) << endl;

    return 0;
}
```

   (a) Is the `square` function declared, defined, or both?

   (b) List every parameter and argument that appears in the code.

   (c) Write down the function signature of the `square` function.

2. (a) Is the function comment in the following code problematic in any way? Does your IDE say anything about it?

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

/**
 This function squares a double.
 @param x : the double to square
 @return x^2
*/
double square(double y) {
    return y * y;
}

int main() {
    double d = 3.0;
    cout << d << "␣squared␣is␣" << square(d) << endl;

    return 0;
}
```

(b) Does the following code encounter a build error, a runtime error, undefined behavior, or does it build and execute successfully on all compilers?

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

/**
 This function squares a double.
 @param x : the double to square
 @return x^2
*/
double square(double x);
double square(double x);
double square(double);

int main() {
    double d = 3.0;
    cout << d << "␣squared␣is␣" << square(d) << endl;

    return 0;
}

double square(double x) {
    return x * x;
}
```

(c) Does the following code encounter a build error, a runtime error, undefined behavior, or does it build and execute successfully on all compilers?

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

int main() {
    double d = 3.0;
    cout << d << "␣squared␣is␣" << square(d) << endl;

    return 0;
}

/**
 This function squares a double.
 @param x : the double to square
 @return x^2
*/
double square(double x) {
    return x * x;
}
```

3. The following code builds and runs without error. Write down the output after execution.

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

int f(int i) {
    cout << "f␣";
    return i * i;
}

int main() {
    f(2);

    int j = f(3);
    int k = f(4);

    cout <<   k  << '␣';
    cout << f(5) << endl;

    return 0;
}
```

4. The following code builds and runs without error. Write down the output after execution.

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

int f(int i, int j, int k) {
    return i * (j - k);
}

int main() {
    int m = 3;
    int k = 5;
    int j = 10;
    int i = 2;

    i = f(j, k, m);

    cout << i << endl;
    return 0;
}
```

5. (a) The following code builds and runs without error. Write down the output after execution.

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

void f(int i) {
    ++i;
}

int main() {
    int i = 0;

    f(i);

    cout << i << endl;

    return 0;
}
```

(b) The following code builds and runs without error. Write down the output after execution.

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

void f(int i, int j) {
    int tmp = i;
    i = j;
    j = tmp;

    cout << i << '␣' << j << endl;
}

int main() {
    int i = 0;
    int j = 1;

    cout << i << '␣' << j << endl;

    f(i, j);

    cout << i << '␣' << j << endl;

    return 0;
}
```

10 References

The references in this section are more precisely called "l-value references". There are also "r-value references". However, l- and r-values, and r-value references are not discussed until PIC 10B. For this reason, we will abbreviate "l-value references" to "references" for the forseeable future.

10.1 Motivating references

The standard provides a procedure called `std::swap` in the `<utility>` header. Example 9.4.3 demonstrated `std::swap`'s behavior. Ignoring all whitespace, line 18 printed `0` followed by `1`, and line 22 printed `1` followed by `0`. This shows `std::swap(i, j)` managed to swap the values of the variables `i` and `j`. Naively, one might think that this is easy to accomplish.

Example 10.1.1. The output of the code below is as follows.

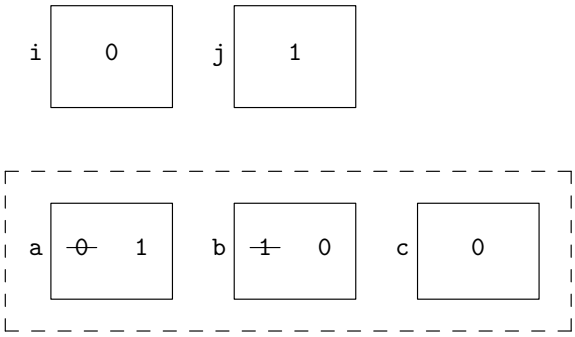
```
1  0 1
2  0 1
```

This shows that `bad_swap` fails to do what `std::swap` does.

```
1  #include <iostream>
2  using namespace std;
3
4  void bad_swap(int a, int b) {
5      int c = a;
6      a = b;
7      b = c;
8  }
9
10 int main() {
11     int i = 0;
12     int j = 1;
13
14     cout << i << '␣' << j << endl;
15
16     bad_swap(i, j);
17
18     cout << i << '␣' << j << endl;
19
20     return 0;
21 }
```

It is good to fully understand why `bad_swap` fails to swap the values of `i` and `j`. A picture is worth

a thousand words.



- The values of the arguments `i` and `j` are copied to the parameters `a` and `b` (contained within the function scope).
- The code within the function body of `bad_swap` successfully swaps the values of `a` and `b`.
- However, the successful swapping of `a` and `b`'s values has nothing to do with `i` and `j`'s values which are left unchanged throughout the execution of the function body's instructions.
- We need a way for the assignments involving `a` and `b` to affect the values of `i` and `j`. References are the answer.

10.2 Introducing references

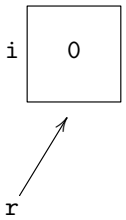
In the following code the type of `r` is not `int`, it is `int&` which means "reference to an `int`". `int&` is often said aloud as "`int` ref".

```
1  int main() {
2      int  i = 0;
3      int& r = i;
4
5      return 0;
6  }
```

So far we have used identifiers (names) to label a storage location for some amount of data.

- A variable of type `int` labels a storage location that can store integers in a certain range using 4 bytes.
- A variable of type `double` labels a storage location that stores numbers with a fractional part of a specific form using 8 bytes.
- A variable of type `char` labels a storage location that can store ASCII characters using 1 byte.

Definition 10.2.1. A *reference* does not store its own data. Instead, it refers to already existing data. It can be helpful to visualize references using arrows. A useful picture for the code above is as follows.



Remark 10.2.2. Some readers of this document may have heard of "pointers". If you have not, you can safely ignore this remark. For those who have, this remark is written to highlight that the concepts are related but distinct. A reference *is* a `const` pointer which is automatically dereferenced. To highlight that the two concepts are distinct, I will *never* draw pointers using arrows. My reasons for this will be explained further once we have addressed pointers in section 17.

10.3 Initializing references and making assignments with references

We have just drawn a reference as an arrow to a storage location. The following definition continues definition 10.2.1...

Definition 10.3.1. A reference can only ever refer to *one* storage location. Once initialized, it will never refer to a second storage location. Said another way, if you draw the reference as an arrow, the arrow is always going to start and end at the same place.

The arrow representing a reference never changing sounds similar to a `const` variable never changing its value. Moreover, there is a build error to be aware of that is very similar to 5.4.1.

Build Error 10.3.2. *Defining a reference without initializing it will result in a build error.*

A reference is designed to refer to a storage location. One is not allowed to ask where literals like `4` (an `int` literal), `3.2` (a `double` literal), `'1'` (a `char` literal), `false` (a `bool` literal) are stored. This is behind the reason for the following build error, but to fully understand it requires a discussion of `l`- and `r`-values, a PIC 10B topic.

Build Error 10.3.3. *The following lines of code give build errors.*

```
1  int main() {
2      int&    ri =   4;
3      double& rd =  3.2;
4      char&   rc =  '1';
5      bool&   rb = false;
6
7      return 0;
8  }
```

The following example addresses how assignments work with references.

Example 10.3.4. The output of the following code has been indicated in the comments.

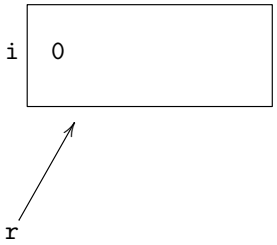
```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       // int& r;       // build error
6       // int& r = 0;   // build error
7
8       int  i = 0;
9       int& r = i;
10
11      cout << i << '␣' << r << endl;               // 0 0
12
13      i = 1;
14      cout << i << '␣' << r << endl;               // 1 1
15
16      r = 2;
17      cout << i << '␣' << r << endl;               // 2 2
18
19      int j = 3;
20      cout << i << '␣' << r << '␣' << j << endl;   // 2 2 3
21
22      r = j;
23      cout << i << '␣' << r << '␣' << j << endl;   // 3 3 3
24
25      i = 4;
26      cout << i << '␣' << r << '␣' << j << endl;   // 4 4 3
27
28      j = r;
29      cout << i << '␣' << r << '␣' << j << endl;   // 4 4 4
30
31      j = 5;
32      cout << i << '␣' << r << '␣' << j << endl;   // 4 4 5
33
34      return 0;
35  }
```

Here is a "video" showing how the code executes.

- On line 8 an `int` called `i` is declared, defined, and initialized with value `0`. On line 9 an `int&`, a reference to an `int`, is declared and made to reference the location that `i` labels. The picture by line 11 is as follows. Printing `r` simply prints the value in the box labelled by `i`. **Follow**

the arrow!

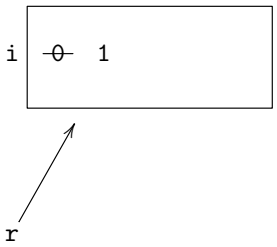
Line 11:



- Line 13 updates `i`'s value to `1`.

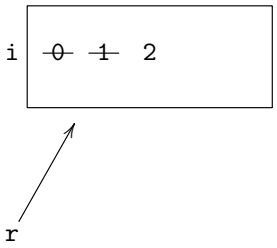
  Because printing `r` prints the value in the box labelled by `i`, both `i` and `r` "see" this change.

Line 14:



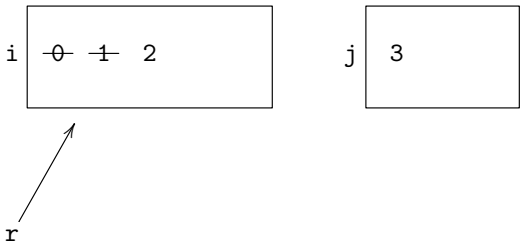
- On line 16, the assignment `r = 2` has the same effect as `i = 2` would have done, updating `i`'s value to `2`. Again, both `i` and `r` "see" this change. To see what value should be changed by `r = 2`, **follow the arrow!**

Line 17:



- On line 19 an `int` called `j` is declared, defined, and initialized with value `3`.

Line 20:

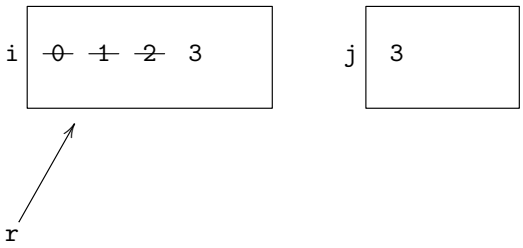


- Line 22 says `r = j`.
  - Remember, **if you draw a reference as an arrow, the arrow is always going to start and end at the same place.**

– Someone who has forgotten this might be tempted, to make `r`'s arrow refer to `j`'s storage location, but because of what we just recalled, **this is not an option.**

On line 22, the assignment `r = j` has the same effect as `i = j` would have done, updating `i`'s value to `3`. Again, both `i` and `r` "see" this change. To see what value should be changed by `r = j`, **follow the arrow!**

Line 23:

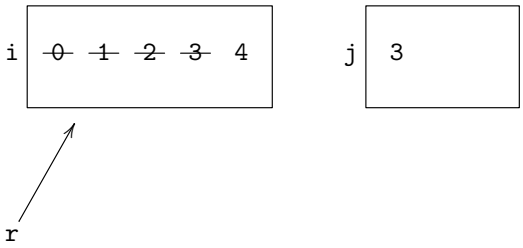


- Line 25 updates `i`'s value to `4`.

  Because printing `r` prints the value in the box labelled by `i`, both `i` and `r` "see" this change.

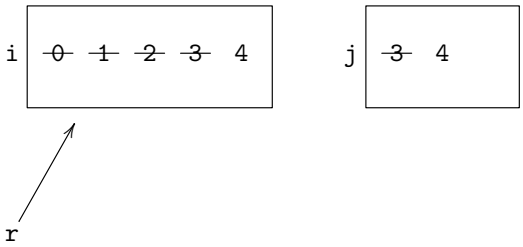
  This line emphasizes once more that `r` is still referring to the location that `i` labels. **It always will!**

Line 26:



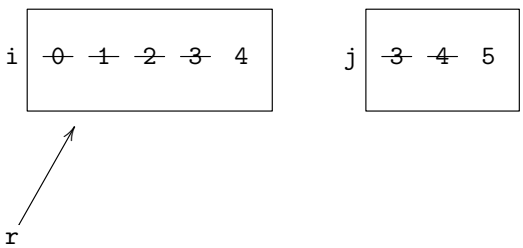
- On line 28, the assignment `j = r` has the same effect as `j = i` would have done, updating `j`'s value to `4`. To see what value should be assigned to `j` when we write `j = r`, **follow the arrow!**

Line 29:



- Line 31 updates `j`'s value to `5`.

Line 32:



This example was a very long way to explain that after declaring a reference to an `int` called `r` and making it reference the location that `i` labels (by writing `int& r = i`), any line involving `r` will behave like we wrote `i` instead.

In general, however, the situation is likely to be more complicated than this because a reference might reference a location labelled by a variable in another scope. In this case, using the variable in the other scope directly might give a build error. This is exactly the situation that occurs when references are used for a function's parameters. In this case, **follow the arrow(s)!**

10.4 Back to the motivating example

We can change the `bad_swap` of example 10.1.1 to `good_swap` by replacing `bad` by `good`(!) and typing two ampersands.

Example 10.4.1. The output of the code below is as follows.

```
0 1
1 0
```

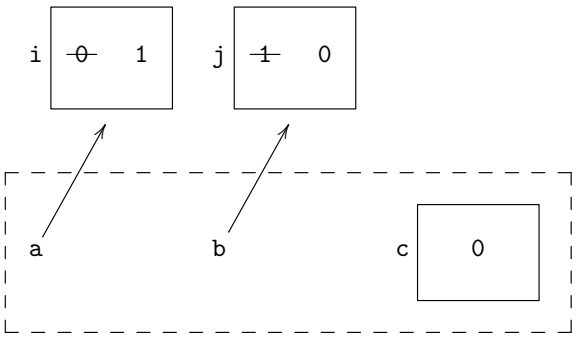
This shows that `good_swap` succeeds at doing what `std::swap` does, well, at least for `int`s. (`std::swap` is still superior in a number of ways that motivate content in PIC 10B and PIC 10C.)

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

void good_swap(int& a, int& b) {
    int c = a;
    a = b;
    b = c;
}

int main() {
    int i = 0;
    int j = 1;

    cout << i << '␣' << j << endl;

    good_swap(i, j);

    cout << i << '␣' << j << endl;

    return 0;
}
```

Because this code uses nice distinct names for the function parameters (`a` and `b`), the variable declared in the function body (`c`), and the function arguments (`i` and `j`), considering the following code gives one way to think about why this attempt at swapping succeeded.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int i = 0;
6       int j = 1;
7
8       cout << i << '␣' << j << endl;
9
10      // good_swap(i, j);
11      {
12          int& a = i;
13          int& b = j;
14
15          int c = a;
16          a = b;
17          b = c;
18      }
19
20      cout << i << '␣' << j << endl;
21
22      return 0;
23  }
```

Because our pictures are not highlighting the difference between a scope introduced by braces and a function scope, drawing the picture for this code will produce the same picture as drawing one for the function call directly.



When `good_swap` is called...

- A function scope is introduced.

  This starts off containing the two parameters `a` and `b`, but **this time they are references.**

- In this situation, assigning the arguments to parameters tells the parameters what location they should reference. The creation of the parameters amounts to the following two lines.

```
12          int& a = i;
13          int& b = j;
```

- The function body executes.
  - A variable called `c` is declared and defined within the function scope.
    It is initialized using the value that `a` references. **Follow the arrow!** That value is `0`.
  - `a = b` executes.
    Both sides of this assignment are references. We have to **follow two arrows.**
    What is updated? **Follow `a`'s arrow.** `main`'s variable `i` is updated.
    What value is it updated with? **Follow `b`'s arrow.** `main`'s variable `j` stores `1`.
    Summarizing, `main`'s variable `i` is updated with the value `1`.
  - `b = c` executes.
    What is updated? **Follow `b`'s arrow.** `main`'s variable `j` is updated.
    What value is it updated with? The function scope's variable `c` stores `0`.
- There is no return value to deal with because the return type is `void`.
- The function scope is destroyed along with `a`, `b`, and `c`. They served their purpose: the values of `i` and `j` have been successfully swapped.

10.5 References to `const`

This content becomes useful in the next section and you may want to skip it until it is relevant.

Definition 10.5.1. In the following code `rc` is said to be a *reference to `const`*.

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

int main() {
    string s(1000000, '!');
    const string& rc = s;

    return 0;
}
```

As the code in the definition shows, a reference to `const` can reference a non-`const` entity. However, the reference will treat the referenced location like a storage location for a `const` variable. Although a non-`const` variable's value can change, the reference will not involve itself in such activities. The following build error feels very similar to 5.4.2.

Build Error 10.5.2. *Assigning to a reference to `const` fundamental type after its initialization gives a build error. The same is usually true for reference to `const` class types.*

There is also the following build error which is explained in the subsequent example.

Build Error 10.5.3. *A reference which is not a reference to* `const` *is not allowed to reference a* `const` *variable. Attempting to do this will result in a build error.*

Example 10.5.4. The output of the following code has been indicated in the comments.

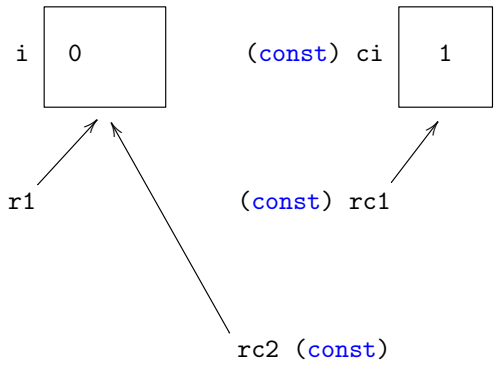
```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int         i = 0;
6       const int ci = 1;
7
8   //  ci = 2;         // build error
9
10      int& r1 =  i;
11  //  int& r2 = ci;   // build error
12
13      const int& rc1 = ci;
14      const int& rc2 =  i;
15
16  //  rc1 = 3;        // build error
17  //  rc2 = 4;        // build error
18
19      cout << i << '␣' << r1 << '␣' << rc2 << endl;   // 0 0 0
20
21      i = 5;
22      cout << i << '␣' << r1 << '␣' << rc2 << endl;   // 5 5 5
23
24      r1 = 6;
25      cout << i << '␣' << r1 << '␣' << rc2 << endl;   // 6 6 6
26
27      return 0;
28  }
```

- Line 5 and 6 declare, define, and initialize two `int`s, the second of which is `const`.
- Uncommenting line 8 would give an example of build error 5.4.2.
- Line 10 declares a reference to an `int` called `r1`, and it references `i`.
- Uncommenting line 11 would give an example of build error 10.5.3.

  It is good to think about why this build error is sensible. If `r2` were allowed to reference `ci`, then, because `r2` is an `int&` and not a `const int&`, `r2` would enable changing `ci`'s value. This would break the contract that we entered when we declared `ci const` on line 6. This build error, like most build errors regarding `const`, protects us from our own stupidity!
- On the other hand, it is completely fine for a reference to `const` like `rc1` (declared on line 13) to reference `ci` because assignments to `rc1` are disallowed: uncommenting line 16 would give an example of build error 10.5.2.

- The rest of the example is centered around `rc2` which is declared on line 14.

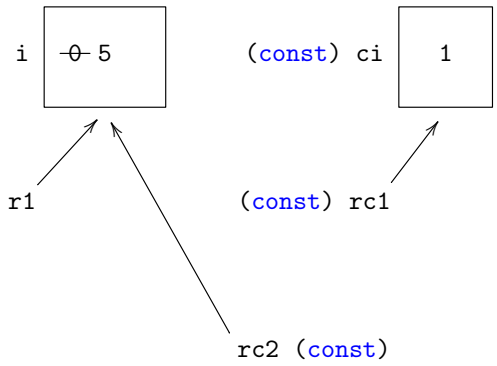
  Here is the picture so far...



  Line 19 prints out `i`, `r1`, `rc2`, all of which access `0` either directly or through an arrow.

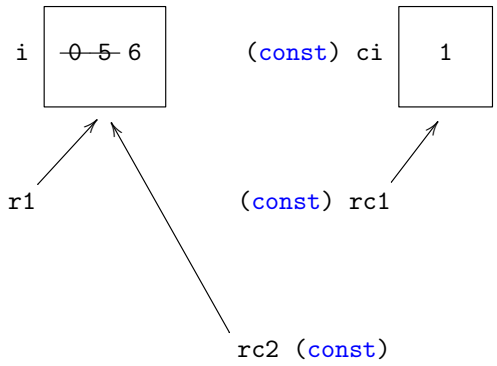
  The code in the example does not bother printing out `ci` and `rc1` because they will always access the value `1`; everything involved is `const`-qualified.

  `i` is not `const`, so line 21 is legal and updates `i`'s value to `5`.



  Line 22 prints out `i`, `r1`, `rc2`, all of which access `5` either directly or through an arrow. Notice that printing `rc2` previously gave `0` and is now giving `5`. The reference can see changes to `i`'s value; it just refuses to participate in make such changes: uncommenting line 17 would give an example of build error 10.5.2.

  `r1` is not a reference to `const`, so line 24 is legal and updates `i`'s value to `6`.



Line 25 prints out `i`, `r1`, `rc2`, all of which access `6` either directly or through an arrow. Again, `rc2` "sees" the change to `i`'s value even though it refuses to participate in make such changes itself.

10.6 Review questions

1. (a) Write down the output from executing the following code.

```
1   #include <iostream>
2   using namespace std;
3
4
5   int f(int& a, int b, int& c) {
6       cout << a << '␣' << b << '␣' << c << endl;
7
8       a  +=  1 + c;
9       b  *=  2;
10
11      cout << a << '␣' << b << '␣' << c << endl;
12
13      return a;
14  }
15
16
17  int main() {
18      int a = 20; int b = 30; int c = 40;
19
20      cout << a << '␣' << b << '␣' << c << endl;
21
22      f(a, b, c);
23
24      cout << a << '␣' << b << '␣' << c << endl;
25
26      return 0;
27  }
```

(b) Write down the output from executing the following code.

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


int f(int& a, int b, int& c) {
    cout << a << '␣' << b << '␣' << c << endl;

    a  +=  1 + c;
    b  *=  2;

    cout << a << '␣' << b << '␣' << c << endl;

    return a;
}


int main() {
    int a = 20; int b = 30; int c = 40;

    cout << a << '␣' << b << '␣' << c << endl;

    f(c, b, a);

    cout << a << '␣' << b << '␣' << c << endl;

    return 0;
}
```

(c) Write down the output from executing the following code.

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


int f(int& a, int b, int& c) {
    cout << a << '␣' << b << '␣' << c << endl;

    a  +=  1 + c;
    b  *=  2;

    cout << a << '␣' << b << '␣' << c << endl;

    return a;
}


```

```
17      int main() {
18          int a = 20; int b = 30; int c = 40;
19
20          cout << a << '␣' << b << '␣' << c << endl;
21
22          f(b, b, b);
23
24          cout << a << '␣' << b << '␣' << c << endl;
25
26          return 0;
27      }
```

2. (a) Does the following code encounter a build error, a runtime error, undefined behavior, or does it build and execute successfully on all compilers?
If it builds and executes successfully on all compilers, write down the output.

```
1       #include <iostream>
2       using namespace std;
3
4       void f(int& r) {
5           ++r;
6       }
7
8       int main() {
9           int i = 0;
10
11          f(i);
12
13          cout << i << endl;
14
15          return 0;
16      }
```

(b) Does the following code encounter a build error, a runtime error, undefined behavior, or does it build and execute successfully on all compilers?
If it builds and executes successfully on all compilers, write down the output.

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

void f(int& r) {
    ++r;
}

int main() {
    f(0);

    cout << 0 << endl;

    return 0;
}
```

(c) The following code produces a build error.

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

void f(int& r) {
    ++r;
}

int main() {
    double d = 0.0;

    f(d);

    cout << d << endl;

    return 0;
}
```

By thinking about the argument and the parameter, can you provide another build error that is at the heart of this one? Your answer should be two lines of code that would go inside the following code to produce a build error. Note that this build error has not been mentioned previously, so you will learn something new here.

```
int main() {
    // Line 1
    // Line 2
    return 0;
}
```

3. Does the following code encounter a build error, a runtime error, undefined behavior, or does it build and execute successfully on all compilers?

   If it builds and executes successfully on all compilers, write down the output.

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

int main() {
          int    i = 0;
    const int    j = 1;
          int& r1 = i;
    const int& r2 = r1;

    cout << i << '␣' << r1 << '␣' << r2 << '␣' << j << endl;

    r1 = j;
    cout << i << '␣' << r1 << '␣' << r2 << '␣' << j << endl;

    i = 2;
    cout << i << '␣' << r1 << '␣' << r2 << '␣' << j << endl;

    return 0;
}
```

11 Vectors (the `std::vector` class template)

11.1 The snippet

See the vectors snippet. It documents...

- Default constructing `std::vector<T>` for various types `T` to create empty vectors.
- The `size`, `push_back`, and `pop_back` member functions.
- `operator[]` and `at`. The following is simpler than 6.2.2.

  Undefined Behavior 11.1.1. *When `v` is a `std::vector<T>` (for some type `T`), typing `v[pos]` gives undefined behavior if `pos >= v.size()`.*

  Runtime Error 11.1.2. *When `v` is a `std::vector<T>` (for some type `T`) and `pos >= v.size()`, typing `v.at(pos)` produces a runtime error.*

- Constructing a `std::vector<T>` using...
  - a `size_t n` to produce a vector containing `n` value-initialized elements (zero in the case of fundamental types, default-constructed instances in the case of class types).
  - a `size_t n` and a `T t` to produce a vector containing `n` copies of `t`.
  - a `std::initializer_list<T>`, that is, some braces `{}` containing a load of `T`s, to produce a vector containing exactly those `T`s.
- The `empty` and `clear` member functions.
- The `resize` and `reserve` member functions.

  The first argument of `resize` is a `size_t`, the new size. The optional second argument can be used to specify the value to use when new elements are required for the resize. If the second argument is not used, new elements will be value-initialized (zero in the case of fundamental types, default-constructed instances in the case of class types).

11.2 A little more

`std::vector` is a class template which means we have to specify a type to get a class. For any type `T`, `std::vector<T>` is a class. For example, all of the following are classes.

- `std::vector<int>`
- `std::vector<double>`
- `std::vector<std::string>`
- `std::vector<std::vector<int>>`

An instance of `std::vector<int>` serves as an adaptive container for `int`s, one which can grow and shrink. It is impossible for a container to have an infinite capacity because this would require infinite memory. Instead, at any moment in time, an instance of `std::vector<int>` takes charge of some finite amount of memory enabling it to store some number of `int`s. If we ask to store more `int`s than that number (the capacity), then the `std::vector<int>` will automatically find a larger chunk of memory and move its existing elements to there so that it is ready to store even more elements in the future. The location of the memory where the `int`s are stored is called *the heap*. Managing memory on the heap is a central topic in PIC 10B, and in that class, I develop useful pictures for discussing that memory management. For now, it suffices to use inaccurate pictures. The story is the same for types `T` other than `int`.

11.3 Printing vectors, another reason for references

Unless we define how to do this ourselves, we cannot `cout` vectors. Said more precisely, whatever type `T` we pick, there is not a standard definition of `operator<<` which accepts a `std::vector<T>`. One can wonder, "why?" In the past, I have conjectured that the C++ committee did not want to make a choice about how vectors are printed because there are too many choices one could make.

- Print the elements separated by a space.
- Print the elements separated by a comma and a space.
- Print the elements separated by a comma and no space.
- Make one of the choices above; also include a symbol like `[` to start the vector and a symbol like `]` to end the vector.
- Consider other symbols for starting and ending the vector like `(` and `)`, `<` and `>`, or `{` and `}`.

Overloading `operator<<` so that we can `cout` vectors is a PIC 10B topic, and so, instead, we will content ourselves with writing a function called `print` which makes one of the choices above.

Example 11.3.1. Consider the following code.

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


void print(vector<int> v) {
    if (v.size() == 0) {
        cout << "{␣}" << endl;
        return;
    }

    cout << "{␣";

    for (size_t i = 0, N = v.size() - 1; i < N; ++i) {
        cout << v[i] << ",␣";
    }
    cout << v[v.size() - 1] << "␣}" << endl;
}


int main() {
    vector<int> w;  print(w);
    w.push_back(0); print(w);
    w.push_back(1); print(w);
    w.push_back(2); print(w);

    return 0;
}
```

It builds and runs without error to produce the following output.

```
{ }
{ 0 }
{ 0, 1 }
{ 0, 1, 2 }
```

The code in the previous example seems okay until one considers printing a vector with a large size and the argument-to-parameter assignment. Imagine if, instead, `main` said...

```
int main() {
    vector<int> w(10000, 8);
    print(w);

    return 0;
}
```

The argument-to-parameter assignment for the function call on line 23 is `vector<int> v = w`. This copies `w`'s ten thousand elements to `v` only to discard them after the printing is done. This copying takes some amount of time. It sounds very unreasonable that to print some elements, we copy them first; when you print a text file on your computer, you do not copy the file first, you just print it!

In section 10, we motivated references by trying to accomplish something similar to the `std::swap` function which manages to mutate the values of the arguments given to it. The example just given provides a second reason to use references: **to avoid the needless copying of data.**

Example 11.3.2. Consider the following code which builds and runs to print ten thousand `8`s.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5
6  void print(vector<int>& v) {
7      if (v.size() == 0) {
8          cout << "{␣}" << endl;
9          return;
10     }
11
12     cout << "{␣";
13
14     for (size_t i = 0, N = v.size() - 1; i < N; ++i) {
15         cout << v[i] << ",␣";
16     }
17     cout << v[v.size() - 1] << "␣}" << endl;
18 }
19
20
21 int main() {
22     vector<int> w(10000, 8);
23     print(w);
24
25     return 0;
26 }
```

Because the type of `print`'s parameter is `vector<int>&` rather than `vector<int>`, the argument-to-parameter assignment is now `vector<int>& v = w` which does not involve any copying of data.

The code in the previous example seems okay until one considers printing a `const` vector. Imagine if, instead, `main` said...

```
21 int main() {
22     const vector<int> w(10000, 8);
23     print(w);
24
25     return 0;
26 }
```

If the function call on line 23 were to use the `print` function defined on lines 6 to 18, the argument-to-parameter assignment would be `vector<int>& v = w`. This would give an example of build error 10.5.3: `w` is marked `const`, but `v` is not a reference to `const`. Because of this, the `print` function defined on lines 6 to 18 is not considered as an option for making sense of the function call on line 23 by any compiler. We should certainly be able to print `const` vectors and so this tells us that printing vectors using a parameter which is a reference that is not a reference to `const` is incorrect. Using a reference to `const` solves all of the previously mentioned problems.

Example 11.3.3. Consider the following code which builds and runs to print ten thousand `1`s and twenty thousand `2`s.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5
6  void print(const vector<int>& v) {
7      if (v.size() == 0) {
8          cout << "{␣}" << endl;
9          return;
10     }
11
12     cout << "{␣";
13
14     for (size_t i = 0, N = v.size() - 1; i < N; ++i) {
15         cout << v[i] << ",␣";
16     }
17     cout << v[v.size() - 1] << "␣}" << endl;
18 }
19
20
21 int main() {
22           vector<int>  w(10000, 1); print( w);
23     const vector<int> cw(20000, 2); print(cw);
24
25     return 0;
26 }
```

- The argument-to-parameter assignments in the example are now...
  - `const vector<int>& v =  w;`
  - `const vector<int>& v = cw;`
- For `print`, the argument-to-parameter assignment will never involve the copying of data.
- The function `print` can print non-`const` and `const` vectors.
- The function signature informs anyone considering using the `print` function of the previous two points *and* reassures them that calling `print(w)` will not mutate `w`.

Remark 11.3.4. It can be useful to think about references to `const` like giving someone access to a Google Doc with read-only permissions. If you want your friend to be able to print one of your Google Docs, you should give them exactly these permissions. If you send them a copy of your Google Doc, then you have created needless extra files in the cyber-universe and if you give them write permissions (like a regular reference), they might accidentally delete your hard work.

11.4 Review questions

1. The following code builds and runs without error. What is the output?

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

int main() {
    vector<int> v = { -2, 4, 6, 8 };

    v.pop_back();
    v.push_back(3);
    v.push_back(9);
    v.push_back(3);
    v.push_back(9);

    v.resize(5);
    v.resize(8, 11);
    v.resize(10);

    for (size_t i = 0, N = v.size(); i < N; ++i) {
        cout << v[i] << '␣';
    }
    cout << boolalpha << v.empty() << endl;

    return 0;
}
```

2. (a) The following code builds and runs without error.

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

void print(vector<int> v) {
    for (size_t i = 0, N = v.size(); i < N; ++i) {
        cout << v[i] << '␣';
    }
    cout << endl;
}

int main() {
    vector<int> v(2000, 8);
    print(v);

    return 0;
}
```

- How many `int`s are printed?
- Approximately how many `int`s are created during the code execution?

(b) The following code builds and runs without error.

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

void print(const vector<int>& v) {
    for (size_t i = 0, N = v.size(); i < N; ++i) {
        cout << v[i] << '␣';
    }
    cout << endl;
}

int main() {
    vector<int> v(2000, 8);
    print(v);

    return 0;
}
```

- How many `int`s are printed?
- Approximately how many `int`s are created during the code execution?

12 How to choose parameter types

12.1 Choosing between passing by value and by reference (to `const`)

Given some C++ type `T` (that does not involve references or `const`), we have now seen three different ways that it could be involved in a parameter's type.

- When we pass an argument by value, the corresponding parameter type will be `T`.
- When we pass an argument by reference, the corresponding parameter type will be `T&`.
- When we pass an argument by reference to `const`, the corresponding parameter type will be `const T&`.

We choose between these parameter types by answering two questions.

1. Do we want to mutate the value of the argument?
2. Do we want to avoid copying the argument?

How do the answers to these questions help us?

1. (a) If the answer to the first question is "yes", then the parameter must have type `T&`.
   (b) If the answer to the first question is "no", then we need to choose between `T` and `const T&`.

2. (a) If the answer to the second question is "yes", then we choose between `T&` and `const T&`.
   (b) Suppose the answer to the second question is "no".
   - In most cases this is likely to mean, "it really does not matter to me whether a copy is made or not". Then all choices are acceptable and `T` is the simplest of them.
   - If the response means, "In fact, I really want a copy," then `T` can accomplish this, or this person can make a copy themselves within the function body.

Putting the two answers together can give us some useful control flow.

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

int main() {
    bool mutate_argument = false;  // initialize according to situation
    bool avoid_copying   = false;  // initialize according to situation

    if (mutate_argument) {
        cout << "T&" << endl;
    }
    else if (avoid_copying) {
        cout << "const␣T&" << endl;
    }
    else {
        cout << "T" << endl;
    }

    return 0;
}
```

What has been said so far is helpful, but it begs another question: when should one avoid copying and when should one not worry about a copy being made? This frequently corresponds to whether `T` is a fundamental type or a class type. Fundamental types always use a small amount of memory, between 1 and 8 bytes. A `std::string`, however, may store the entire works of Shakespeare, more than 3 million characters. A `std::vector<int>` could store many, many `int`s in a useful bit of code. Therefore, more often than not, when one has a class type, one tries to avoid a copy being made. Therefore, in many situations, but not all, the following control flow does a good job of telling you what to do.

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

int main() {
    bool mutate_argument = false;  // initialize according to situation
    bool class_type      = false;  // initialize according to situation

    if (mutate_argument) {
        cout << "T&" << endl;
    }
    else if (class_type) {
        cout << "const␣T&" << endl;
    }
    else {
        cout << "T" << endl;
    }

    return 0;
}
```

Remark 12.1.1. There are situations when one has a class type but it is reasonable to pass by value. For example, if an algorithm amounts to copy and edit, then it is useful to let the parameter take care of the copying. This can result in efficiency gains in some situations like in the following code.

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

void f1(string s) {
    if (s.length() > 0) {
        s[0] = 'A';
    }
    cout << s << endl;
}

void f2(const string& s) {
    string copy = s;

    if (copy.length() > 0) {
        copy[0] = 'A';
    }
    cout << copy << endl;
}

int main() {
    f1("a␣string");
    f2("a␣string");

    return 0;
}
```

- When `f1("a␣string")` is called, the parameter `s` is constructed by using the argument string literal `"a␣string"`, and then the parameter is edited and printed.
- When `f2("a␣string")` is called, a `std::string` is constructed using the string literal argument `"a␣string"`, it is referenced by the parameter `s`, then copied to `copy`, and then `copy` is edited and printed.

The first function call does not actually involve a copy, just the construction of a `std::string` from a string literal and is more efficient than the second function call. Many people say that passing by value amounts to copying an argument, but when the parameter type is a class type, the parameter needs constructing, and classes can have many constructors, not just a copy constructor.

In section 15, you will learn more about constructors.

If you take PIC 10B, you will learn about copy constructors and move constructors.

This remark is more advanced and can safely be ignored at this stage.

Remark 12.1.2. You may have wondered about a fourth parameter type: `const T`. This parameter type is almost never used. In fact, the following code will give a build error saying that line 2 gives a redefinition of the `f` on line 1, so, to some extent, compilers ignore `const` on a parameter that is not a reference.

```
1  void f(      int i) {}
2  void f(const int i) {}
3
4  int main() {
5      return 0;
6  }
```

The only time I have seen this used is when someone defining a function wishes, for some reason, to make sure they do not change the value of a parameter, but this is not something that the rest of the world should see.

```
1  // DOCUMENTATION
2  // Function declaration...
3  void f(int i);
4
5  int main() {
6      return 0;
7  }
8
9  // IMPLEMENTATION
10 // Function definition...
11 void f(const int i) {
12     /*
13      I am implementating this function
14      and I wish to protect myself against
15      accidentally changing the parameter 'i'.
16
17      I expect no one will ever read this.
18     */
19     // ++i;  // Build error.
20 }
```

You can safely ignore this remark and never use a `const` non-reference parameter.

Example 12.1.3. The following code builds and runs to produce an output of `6`.

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

int sum(const vector<int>& v) {
    int sum = 0;

    for (size_t i = 0, N = v.size(); i < N; ++i) {
        sum += v[i];
    }
    return sum;
}

void print_sum(const vector<int>& v) {
    cout << sum(v) << endl;
}

int main() {
    vector<int> w { 1, 2, 3 };
    print_sum(w);

    return 0;
}
```

The functions `sum` and `print_sum` correctly use references to `const` to avoid copying their arguments and to indicate that their arguments will not be mutated. Using `const` correctly is referred to as being "`const` correct". Being `const` correct is important. To see why, suppose that different people are writing these functions.

- Suppose person 1 writes `sum` and is not `const` correct.
- Suppose person 2 writes `print_sum` and is `const` correct.

What is the result? Deleting the `const` in `sum` creates a build error for `print_sum`, but this is not person 2's fault. Therefore, person 2 becomes angry at person 1 and decides not to work with them anymore. If you want to work with others, you need to be `const` correct!

12.2 Review questions

1. You wish to define a function to arrange the elements in a `std::vector<int>` in ascending order. If `v` is a `std::vector<int>`, after calling `sort(v)`, the elements in `v` should be in ascending order. What is the correct choice for the parameter type of `sort`?

2. You wish to write a function that can find a value within a `std::vector<double>`. To demonstrate, the output of the following code should be `2 true`.

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

// Pretend there is a good definition of 'find'.

int main() {
    vector<double> v { 0.0, 1.1, 2.2, 3.3 };

    cout <<  boolalpha;
    cout <<  find(v, 2.2)                                 << '␣';
    cout << (find(v, 4.4) == static_cast<size_t>(-1)) << endl;

    return 0;
}
```

   What are the best choices for the parameter types of `find`?

3. The following code produces a build error.

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

int sum(vector<int>& v) {
    int sum = 0;

    for (size_t i = 0, N = v.size(); i < N; ++i) {
        sum += v[i];
    }
    return sum;
}

void print_sum(const vector<int>& w) {
    cout << sum(w) << endl;
}

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

Which of the following is the reason for this build error?

- The parameter of `sum` is called `v`, but the argument passed to it is called `w`.
- The code involves making a reference to `const` reference a variable that is not `const`.
- The argument-to-parameter assignment for the function call is `vector<int>& v = w`. This creates a build error because both `v` and `w` are references.
- The argument-to-parameter assignment for the function call is `vector<int>& v = w`. This creates a build error because `w` is a reference to `const` and `v` is not a reference to `const`.
- If the `sum` function defined in the code above was used to resolve the function call `sum(w)`, the argument-to-parameter assignment would be `vector<int>& v = w`. This is invalid because `w` is a reference to `const` and `v` is not a reference to `const`. For this reason, the `sum` function defined above is not considered, and there is no other matching function.

13 Header file and cpp arrangement

13.1 Introducing the idea

C++ code is often organized into header files, which have the extension `.h` or `.hpp`, and `.cpp` files. If a header file has a corresponding `.cpp` file, then using the extension `.hpp` can emphasize this. I use the extension `.h` only when there is no corresponding `.cpp` file, but some people use `.h` for every header file.

Example 13.1.1. Here is an example where code has been distributed across multiple files. You will have to use your imagination a little bit to think of some awesome audio, image, and video applications! First, the `.hpp` files...

audio.hpp

```
#include <iostream>

/**
 Does an awesome audio thing.
*/
void do_awesome_audio_thing();
```

image.hpp

```
#include <iostream>

/**
 Does an awesome image thing.
*/
void do_awesome_image_thing();
```

video.hpp

```
#include <iostream>
#include "audio.hpp"  // videos need audio
#include "image.hpp"  // videos need images

/**
 Does an awesome video thing.
*/
void do_awesome_video_thing();
```

Next, the .cpp files.

audio.cpp

```
#include "audio.hpp"

void do_awesome_audio_thing() {
    std::cout << "awesome audio thing is being done" << std::endl;
}
```

image.cpp

```
#include "image.hpp"

void do_awesome_image_thing() {
    std::cout << "awesome image thing is being done" << std::endl;
}
```

video.cpp

```
#include "video.hpp"

void do_awesome_video_thing() {
    std::cout << "awesome video thing is being done, ";
    std::cout << "and so..." << std::endl;

    do_awesome_audio_thing();
    do_awesome_image_thing();
}
```

main.cpp

```
#include <iostream>
#include "audio.hpp"
#include "image.hpp"
#include "video.hpp"

int main() {
    do_awesome_audio_thing(); std::cout << std::endl;
    do_awesome_image_thing(); std::cout << std::endl;
    do_awesome_video_thing(); std::cout << std::endl;
    return 0;
}
```

The `#include`s are simpler than you may have imagined. When `#include "audio.hpp"` is written, it is as though the entire contents of that file have been typed. Therefore, `main.cpp` is ...

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

/**
 Does an awesome audio thing.
*/
void do_awesome_audio_thing();

#include <iostream>

/**
 Does an awesome image thing.
*/
void do_awesome_image_thing();

#include <iostream>
#include <iostream>

/**
 Does an awesome audio thing.
*/
void do_awesome_audio_thing(); // videos need audio

#include <iostream>

/**
 Does an awesome image thing.
*/
void do_awesome_image_thing(); // videos need images

/**
 Does an awesome video thing.
*/
void do_awesome_video_thing();

int main() {
    do_awesome_audio_thing(); std::cout << std::endl;
    do_awesome_image_thing(); std::cout << std::endl;
    do_awesome_video_thing(); std::cout << std::endl;
    return 0;
}
```

`do_awesome_audio_thing` ends up being declared twice:

- once due to writing `#include "audio.hpp"` in `main.cpp`
- once due to writing `#include "audio.hpp"` in `video.hpp` and writing `#include "video.hpp"` in `main.cpp`.

Similarly, `do_awesome_image_thing` is declared twice, and `#include <iostream>` is written six times! We saw before that declaring a function multiple times is okay, and this code builds and runs to give the following output.

```
awesome audio thing is being done

awesome image thing is being done

awesome video thing is being done, and so...
awesome audio thing is being done
awesome image thing is being done
```

13.2 How to organize your code

Suppose that you are writing some functions and you wish to organize your function comments, function declarations, and function definitions across files called `fs.hpp` and `fs.cpp`. Header files generally serve as documentation that people will look at and then cpp files provide implementation details that many people wish to ignore. Therefore...

- Function comments and declarations should go in the header file: `fs.hpp`.
- Function definitions should go in the corresponding `.cpp` file: `fs.cpp`.
- `fs.cpp` file should `#include "fs.hpp"`.
- You should not use any namespaces in `fs.hpp`.

  Using a namespace in a header file will force people who include your header file to use that namespace and they may not want to use that namespace.
- It is okay to use namespaces in `fs.cpp`.

  However, my personal preference is to avoid this and only use namespaces within `main.cpp`.
- I normally place relevant `#include`s within the header file since they provide some amount of documentation (along with the function comments and signatures).

Example 13.2.1. Here is a shorter example.

functions.hpp

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

/**
 Prints the elements of a vector of ints
 separated by a comma and a space and
 enclosed by braces {}.
 @param v : the vector of ints to print
*/
void print(const std::vector<int>& v);
```

functions.cpp

```
#include "functions.hpp"
using namespace std;

void print(const vector<int>& v) {
    if (v.size() == 0) {
        cout << "{␣}" << endl;
        return;
    }

    cout << "{␣";

    for (size_t i = 0, N = v.size() - 1; i < N; ++i) {
        cout << v[i] << ",␣";
    }

    cout << v[v.size() - 1] << "␣}" << endl;
}
```

main.cpp

```
#include "functions.hpp"
using namespace std;

int main() {
    vector<int> w { 1, 2, 3 };
    print(w);

    return 0;
}
```

Remark 13.2.2. This arrangement may lead to questions like, "If `main.cpp` includes `functions.hpp`, how does it know about the function definitions in `functions.cpp`?"

The short answer is that `main.cpp` does **not** know about the definitions in `functions.cpp`!

- When the project is compiled, both `main.cpp` and `functions.cpp` are compiled.

  (`functions.hpp` is not compiled.)

- For these compilations, declarations are necessary for any function that is called.
  - `#include "functions.hpp"` provides the declaration of `print` for `main.cpp`.
  - `#include "functions.hpp"` provides the declaration of `std::cout` and `std::vector<int>` (among other things) for `functions.cpp`.

- After compilation, which produces "object files", linking is done which will fill in any missing definitions. In the previous example...
  - The object file corresponding to `main.cpp` will have a definition for `main` and a missing definition for `print`.
  - The object file corresponding to `functions.cpp` will have a definition for `print`.
  - Because of the declarations, the linker has enough information to stitch these object files together into one executable so that `main` can successfully call `print`.

14 Function Overloading

14.1 Resolving function calls

Function overloading is one of the best parts of C++! First, we note what is not allowed.

Build Error 14.1.1. *Declaring two functions with*

- *the same name*
- *the same parameter types (this means in the same order too)*
- *different return types*

gives a build error.

Example 14.1.2. The following code encounters a build error because there are two functions with the same name `f`, no parameters, and different return types: `int` and `double`.

```
int    f() { return 0;   }
double f() { return 0.0; }

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

Example 14.1.3. The following code encounters a build error because two functions have the same name `f`; two parameters each, the first with type `bool`, the second with type `char`; and different return types: `int` and `double`.

```
int    f(bool b, char c) { return 0;   }
double f(bool l, char r) { return 0.0; }

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

Example 14.1.4. The following code does **not** encounter a build error. The two functions have the same name `f`. They have two parameters each. However, the first parameter of the first `f` has type `char` and the first parameter of the second `f` has type `bool`.

```
int    f(char c, bool b) { return 0;   }
double f(bool l, char r) { return 0.0; }

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

Let that sink in. The last bit of code does **not** encounter a build error.

Definition 14.1.5. It is legal to define functions with the same name and different parameter types. Doing so is called *function overloading*.

Example 14.1.6. The code below builds and runs to give the following output.

```
{ 1, 2 }
(1.1 2.2)
"12"
```

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

void print(const vector<int>& v) {
    cout << "{␣";

    for (size_t i = 0, N = v.size(); i < N; ++i) {
        cout << v[i];
        if (i + 1 < N) { cout << ",␣"; }
    }
    cout << "␣}";
}

void print(const vector<double>& v) {
    cout << '(';

    for (size_t i = 0, N = v.size(); i < N; ++i) {
        cout << v[i];
        if (i + 1 < N) { cout << '␣'; }
    }
    cout << ')';
}

void print(const vector<char>& v) {
    cout << '"';

    for (size_t i = 0, N = v.size(); i < N; ++i) {
        cout << v[i];
    }
    cout << '"';
}

int main() {
    vector<int>    vi { 1 , 2 };  print(vi);  cout << endl;
    vector<double> vd { 1.1, 2.2 };  print(vd);  cout << endl;
    vector<char>   vc { '1', '2' };  print(vc);  cout << endl;

    return 0;
}
```

The previous example raises the question of how function calls are resolved when function overloading is involved. We saw that...

- calling `print` with a `vector<int>` led to `print(const vector<int>&)` being used;
- calling `print` with a `vector<double>` led to `print(const vector<double>&)` being used;
- calling `print` with a `vector<char>` led to `print(const vector<char>&)` being used.

So it also demonstrates that in simple cases what you would want to happen does happen!

The compiler does the job of resolving function calls. CPP reference states...

> In simple terms, the overload whose parameters match the arguments most closely is the one that is called.

It then goes on to explain all the rules in a very, very, very long article. Rather than go into all of those rules, I will provide some examples, and explain which rules determine their behavior.

Example 14.1.7. The following example shows that casting can be involved when a function is called. The fact that line 8 and line 11 produce an output of `1` and `6.1`, respectively, is unsurprising.

```
1  #include <iostream>
2  using namespace std;
3
4  int    int_f(int    i) { return i;       }
5  double dbl_f(double d) { return d + 0.5; }
6
7  int main() {
8      cout << int_f(1)   << endl;  // 1
9      cout << int_f(2.3) << endl;  // 2
10     cout << dbl_f(4)   << endl;  // 4.5
11     cout << dbl_f(5.6) << endl;  // 6.1
12
13     return 0;
14 }
```

- When `int_f(2.3)` is called, `int_f` is the only viable function.

  The parameter-to-argument assignment is `int i = 2.3;` the `double` `2.3` is cast to the `int` `2`.

- When `dbl_f(4)` is called, `dbl_f` is the only viable function.

  The parameter-to-argument assignment is `double d = 4;` the `int` `4` is cast to the `double` `4.0`.

Example 14.1.8. In the following example, there are function overloads called `f`. Each time `f` is called, both overloads are viable candidates because we can cast between `int`s and `double`s.

The output from lines 8 and 9 is unsurprising. In each case, one of `f`'s overloads has parameter types which perfectly match the argument types, and the chosen function's argument-to-parameter assignments involve no casting.

```
1   #include <iostream>
2   using namespace std;
3
4   void f(int a,    int b, double c) { cout << "i␣i␣d" << endl; }
5   void f(int a, double b, double c) { cout << "i␣d␣d" << endl; }
6
7   int main() {
8       f(1,   3,   5.6);   // i i d
9       f(1,   3.4, 5.6);   // i d d
10
11      cout << endl;
12
13      f(1,   3,   5);     // i i d
14      f(1.2, 3.4, 5.6);   // i d d
15
16      return 0;
17  }
```

When one considers the two options for resolving the function call `f(1, 3, 5)` one finds...

- The first parameter type of both overloads (`int`) perfectly matches the argument type (`int`).
- The third parameter type of both overloads (`double`) disagrees with the argument type (`int`), and the argument-to-parameter assignments would both require casting from `int` to `double`.
-
  - The second parameter type of the first overload (`int`) perfectly matches the argument type (`int`).
  - The second parameter type of the second overload (`double`) disagrees with the argument type (`int`), and the argument-to-parameter assignment would require casting from `int` to `double`.

In summary, the first and third positions give a tie, but the second position gives a reason to choose the first overload.

Similarly, for the function call `f(1.2, 3.4, 5.6)`, the first and third positions give a tie, but the second position gives a reason to choose the second overload.

Example 14.1.9. In the following example, there are function overloads called `g`. Each time `g` is called, both overloads are viable candidates because we can cast between `int`s and `double`s.

The output from lines 8 and 9 is unsurprising. In each case, one of `g`'s overloads has parameter types which perfectly match the argument types, and the chosen function's argument-to-parameter assignments involve no casting.

```
1   #include <iostream>
2   using namespace std;
3
4   void g(int    a,    int b) { cout << "i␣i" << endl; }
5   void g(double a, double b) { cout << "d␣d" << endl; }
6
7   int main() {
8       g(1,   3);     // i i
9       g(1.2, 3.4);   // d d
10
11  //  g(1,   3.4);   // build error
12  //  g(1.2, 3);     // build error
13
14      return 0;
15  }
```

There are two ways that the compiler can make sense of the commented function call on line 11.

- `void g(int, int)` would have to cast during the second argument-to-parameter assignment.
- `void g(double, double)` would cast during the first argument-to-parameter assignment.

Neither overload can be regarded as a better match. Because of this, the commented function call on line 11 would give a build error. Similarly, the commented function call on line 12 would give a build error.

Build Error 14.1.10. *When there is not a "best" viable candidate function, a function call gives a build error. If we are to regard one function overload as a "better" match than another function overload, it cannot be "worse" in any argument/parameter position. To regard one function overload as a "better" match than another function overload, it must be equal or "better" in every argument/parameter position and strictly "better" in at least one argument/parameter position.*

Example 14.1.11. The following code encounters a build error.

```
1   #include <iostream>
2   using namespace std;
3
4   void h(int    a, int    b, int    c) { cout << "i␣i␣i" << endl; }
5   void h(double a, double b, double c) { cout << "d␣d␣d" << endl; }
6
7   int main() {
8       h(1, 2, 3.4);
9
10      return 0;
11  }
```

There are two ways that the compiler can make sense of the commented function call on line 8.

- `void h(int, int, int)` would have to cast in the third argument-to-parameter assignment but matches perfectly in the other positions.
- `void h(double, double, double)` would cast in the first and second argument-to-parameter assignments but matches perfectly in the third postion.

Neither overload can be regarded as a better match because both are worse in at least one position.

Example 14.1.12. The code below builds and runs to produce the following output.

```
      vector<int>&
const vector<int>&
```

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

void j(      vector<int>& p) { cout << "␣␣␣␣␣␣vector<int>&" << endl; }
void j(const vector<int>& p) { cout << "const␣vector<int>&" << endl; }

int main() {
          vector<int>  v;   j(v);
    const vector<int> cv;  j(cv);

    return 0;
}
```

- Both overloads are viable functions for the first function call.
  - The first overload would have `vector<int>& p = v` as its argument-to-parameter assignment.
  - The second overload would have `const vector<int>& p = v` as its argument-to-parameter assignment.

  The first feels like a better match and the C++ rules agree with these feelings.
- The second overload is the only viable function for the second function call. The first overload is not viable because `const vector<int> cv; vector<int>& p = cv;` would give a build error; since `cv` is marked `const`, `void j(vector<int>&)` is not a viable function for the function call `j(cw)`.

Remark 14.1.13. Some of the comments after example 11.3.2 can be expressed more consisely by saying if `w` is marked `const`, then `void print(vector<int>&)` is not a viable function for the function call `print(w)`.

14.2 Review questions

1. Consider the following code.

   - If it encounters a build error, try to explain the build error as well as you can.
   - If it builds and executes successfully on all compilers, write down the output.

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

void f()                { cout << '1'      << endl; }
void f(char c)          { cout << '2' << c << endl; }
void f(bool b, char c) { cout << '3' << b << endl; }

int main() {
    f('c', false);

    return 0;
}
```

2. The following code encounters a build error.

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

void f(int  i) { cout << '1' << i << endl; }
void f(char c) { cout << '2' << c << endl; }

int main() {
    f(0.0);

    return 0;
}
```

   What is the reason for this build error?

   - The argument to the function call is a `double` and neither overload of `f` is viable because their parameters are of different types.
   - Both functions are viable and...
     - the first overload is worse in the first position because the argument-to-parameter assignment would involve a casting;
     - the second overload is worse in the first position because the argument-to-parameter assignment would involve a casting.
   - Both functions are viable, the argument-to-parameter assignments both involve casting, and the C++ rules do not regard one casting as "better" than the other.

3. The following code does **not** encounter a build error. This might seem surprising, especially when considering the previous question.

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

void f(int    i) { cout << '1' << i << endl; }
void f(double d) { cout << '2' << d << endl; }

int main() {
    f('X');

    return 0;
}
```

The precise reason for why this function call is resolvable is buried in the CPP reference page that was linked to earlier. However, can you make a sensible guess as to why this function call is resolvable by the compiler?

4. Consider the following code.

   - If it encounters a build error, try to explain the build error as well as you can.
   - If it builds and executes successfully on all compilers, write down the output.

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

void f(const string& s) { cout << '1' << endl; }

void g(      string& s) { cout << '2' << endl; }
void g(const string& s) { cout << '3' << endl; }

int main() {
          string s;
    const string cs;

    f(s);
    f(cs);

    g(s);
    g(cs);

    return 0;
}
```

15 Classes

15.1 Declaring and defining structs, member variables

Definition 15.1.1. In the code below, the first line *declares and defines a struct* called `Rectangle`. This creates a new class type for C++, so that on line 4 and 5 we can default construct instances of `Rectangle`.

```
1  struct Rectangle {};
2
3  int main() {
4      Rectangle r;
5      Rectangle another_r;
6
7      return 0;
8  }
```

Remark 15.1.2. It is possible to separate the declaration and definition of a struct. We will not see a good reason for doing this in PIC 10A, but it can be useful. Typing `struct Rectangle;` would provide a declaration.

It is really important to keep the following in mind while learning about structs (and classes).

- The **definition of a struct** provides a **blueprint** for instances of the struct. It says what instances *will* look like when they are constructed, but it does not create any instances.

  In "real life", when an architect or structural engineer draws a blueprint, a building does not suddenly appear! The building still requires a construction team to build it.

- In order to see the consequences of defining a struct, **one needs to construct instances** of it.

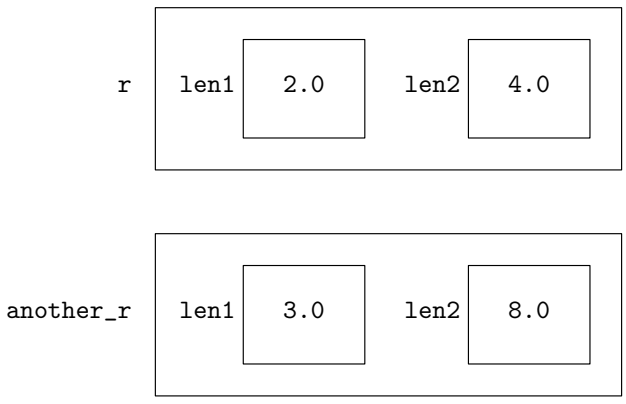
  It is surely the dream of a young architect to have one of their blueprints turned into a real building. Unless one is built, the value of the blueprint may only be that of an abstract idea in the young architect's mind.

Because there is nothing written within the braces on line 1 of the code in the previous definition, the datatype is not very deserving of the name `Rectangle`. What should the blueprint of a `Rectangle` be? Well, rectangles have four sides of two lengths. It would be nice if an instance of a `Rectangle` had two values associated with it describing the side lengths.

Definition 15.1.3. In the code below, the first 4 lines declare and define a struct called `Rectangle`.

```
1   struct Rectangle {
2       double len1;
3       double len2;
4   };
5
6   int main() {
7       Rectangle r;
8       r.len1 = 2;
9       r.len2 = 4;
10
11      Rectangle another_r;
12      another_r.len1 = 3;
13      another_r.len2 = 8;
14
15      return 0;
16  }
```

- Line 2 says that all instances of `Rectangle` will have a *member variable* of type `double` called `len1`. Similarly, line 3 says that all instances of `Rectangle` will have a member variable of type `double` called `len2`.
- Line 7 default constructs an instance of `Rectangle` called `r`.
- Line 8 uses the *member access operator* `.` to access the member variable `len1` belonging to `r`; its value is updated to `2.0`. Similarly, line 9 uses the member access operator `.` to access the member variable `len2` belonging to `r` and its value is updated to `4.0`.
- Lines 11, 12, and 13 construct another instance of `Rectangle` called `another_r` and make sure that the values of its member variables `len1` and `len2` are `3.0` and `8.0`, respectively.
- The following picture clearly shows the four `double`s that have been created, that there are two `len1`s (one belonging to `r` and another belonging to `another_r`), and that there are two `len2`s (one belonging to each instance). Moreover, you can think of the member access operator as "going inside the box". This picture portrays how memory is managed quite accurately.



Example 15.1.4. Here is one of the first function overloads that you learn about in math class.

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

struct Rectangle {
    double len1;
    double len2;
};

struct Square {
    double len;
};

struct Circle {
    double radius;
};

double area(const Rectangle& r) { return           r.len1    * r.len2;   }
double area(const Square&    s) { return           s.len     * s.len;    }
double area(const Circle&    c) { return acos(-1) * c.radius * c.radius; }

int main() {
    Rectangle r;
    r.len1 = 2; r.len2 = 4;

    Square s;
    s.len = 3;

    Circle c;
    c.radius = 1;

    cout << area(r) << '␣';
    cout << area(s) << '␣';
    cout << area(c) << endl;

    return 0;
}
```

The output is `8 9 3.14159` and records the area of a 2×4 rectangle, a 3×3 square, and a circle (the homotopy theorist in me wishes this said "disk") of radius 1.

15.2 Member functions

Example 15.2.1. Although the previous example works quite nicely, consider the following code.

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   using namespace std;
5
6   int main() {
7       vector<int> v { 1, 2, 3 }
8
9       v.resize(0);
10      v.size();
11
12      string s("Hello!");
13
14      s.resize(4);
15      s.size();
16
17      return 0;
18  }
```

- On line 9 we do not not write `resize(v, 0)`.

  On line 10 we do not write `size(v)`.

  There are member functions called `resize` and `size` belonging to the `std::vector<int>` class.

- On line 14 we do not write `resize(s, 4)`.

  On line 15 we do not write `size(s)`.

  There are member functions called `resize` and `size` belonging to the `std::string` class.

- There is not a function overload called `resize`.

- There is also not a function overload called `size`. However, to be entirely honest, there is a function template (function templates are a PIC 10C topic) called `size` which simply calls the relevant member function.

These observations suggest that it would have been better to write member functions called `area` for each of the structs: `Rectangle`, `Square`, and `Circle`.

Example 15.2.2. Here is example 15.1.4 but written using member functions instead of function overloads.

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

struct Rectangle {
    double len1;
    double len2;

    double area() const { return len1 * len2; }
};

struct Square {
    double len;

    double area() const { return len * len; }
};

struct Circle {
    double radius;

    double area() const { return acos(-1) * radius * radius; }
};

int main() {
    Rectangle r;  r.len1   = 2;  r.len2 = 4;
    Square    s;  s.len    = 3;
    Circle    c;  c.radius = 1;

    cout << r.area() << '␣';
    cout << s.area() << '␣';
    cout << c.area() << endl;

    return 0;
}
```

The output is `8 9 3.14159` and records the area of a 2×4 rectangle, a 3×3 square, and a disk of radius 1.

Some aspects of the previous example look reasonable, but there is lots to be confused by on a first encounter. There are three newly-positioned `const`s which we will discuss eventually. The most confusing part, however, is that the function defined "inside of `Rectangle`" seems to have no parameters, and that its `return` value is calculated by multiplying two member variables that are not accessed by using the member access operator.

We need language to talk about this stuff...

Definition 15.2.3. In the following description (as in the other definitions in this document), all newly mentioned terms are in italic.

In the code below, the first eight lines declare and define a struct called `C`. They also constitute the *interface* of the struct. The interface tells us that instances of `C` will have a member variable called `i`. On lines 4 to 7, the interface also declares and defines a *member function* whose full name is `C::f`. Here, `::` is called *the scoping operator* and it is used to specify which struct `f` is a member of. The member function `C::f` has one *explicit parameter* called `j`.

```
1   struct C {
2       int i;
3
4       double f(int j) {
5           int sum = i + j; i *= 10;
6           return sum / 2.0;
7       }
8   };
9
10  int main() {
11      C c; c.i = 3;
12
13      c.f(4);
14
15      return 0;
16  }
```

On line 11 an instance of `C` called `c` is default constructed; the value of its member variable `i` is updated to `3`. On line 13 the member function `C::f` is called with one *explicit argument*: `4`. The instance of `C` before the member access operator, that is, `c`, is called the *implicit argument*.

At the start of the function call, the implicit argument will be assigned to the *implicit parameter*. The implicit parameter refers to an instance of `C`, and so it has access to a member variable `i`. In the function body, on line 5, the `i`s are used to access this member variable. In the body of a member function, one is able to access member variables (and invoke member functions) without needing to type the implicit parameter.

That was a lot of language, and the way that we are allowed to define these member functions probably feels very alien. Just as for free functions (that is, non-member functions), the picture for member functions explains a lot. So let's understand how the slightly expanded code on the next page executes. Hopefully this can clarify some of what was just said.

Example 15.2.4. The following code builds and runs successfully to produce output.

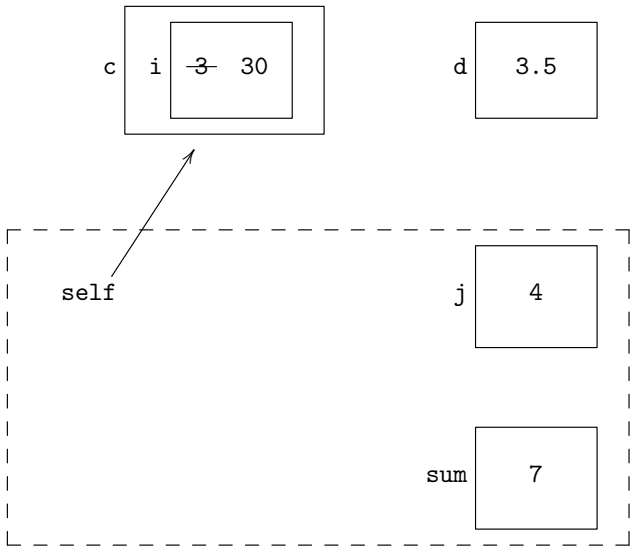
```
1   #include <iostream>
2   using namespace std;
3
4   struct C {
5       int i;
6
7       double f(int j) {
8           int sum = i + j;
9
10          i *= 10;
11
12          return sum / 2.0;
13      }
14  };
15
16  int main() {
17      C c; c.i = 3;
18
19      double d = c.f(4);
20
21      cout << c.i << '␣' << d << endl;
22
23      return 0;
24  }
```

- On line 17 an instance of `C` called `c` is default constructed and the value of its member variable `i` is updated to `3`.
- The following code gives a good way to think about the function call on line 19.

```
19          double d;
20          {
21              // implicit-argument-to-implicit-parameter assignment
22              C& self = c;
23
24              // explicit-argument-to-explicit-parameter assignment
25              int j = 4;
26
27              // 'i' accesses 'self.i'
28              int sum = self.i + j;
29
30              // 'i' accesses 'self.i'
31              self.i *= 10;
32
33              // handling the return value
34              d = sum / 2.0;
35          }
```

At the start of the function call, the implicit argument `c` is assigned to the implicit parameter which I have called `self` and whose type is a reference to `C`. Any line of the function body that uses `i` will make use of `self.i`, the `i` belonging to the instance of `C` that called `C::f`.

The picture is as follows.



The output is `30 3.5`.

Remark 15.2.5. In C++, there is a keyword called `this` whose type is a pointer (section 17). Since we have not spoken about pointers yet, I am currently replacing `this` by `*this` (the dereferenced pointer), and calling it `self`. My use of the word `self` is inspired by Python and it is not a keyword in C++ (or Python, in fact).

We can now return to the `Rectangle` example.

Example 15.2.6. The following code produces an output of `8.8`.

The function call on line 15 is described by the commented lines 17 to 22.

The `const` on line 8 marks `area const`. Marking a member function `const` makes the implicit parameter a reference to `const` like on line 19. This is done because calling `area` on an instance of `Rectangle` should not lead to changing any of the member variables' values.

```
1  #include <iostream>
2  using namespace std;
3
4  struct Rectangle {
5      double len1;
6      double len2;
7
8      double area() const { return len1 * len2; }
9  };
10
11 int main() {
12     Rectangle r;
13     r.len1 = 2; r.len2 = 4.4;
14
15     double a = r.area();
16
17 //  double a;
18 //  {
19 //      const Rectangle& self = r; // implicit arg-to-param assignment
20 //      a = self.len1 * self.len2; // len1 accesses self.len1, i.e. a.len1
21 //                                 // len2 accesses self.len2, i.e. a.len2
22 //  }
23
24     cout << a << endl;
25
26     return 0;
27 }
```

It would probably be good to record what was just said about `const` as a definition, but we will return to `const` after saying something about constructors.

15.3 Constructors

Recall that we have a few handy ways to construct instances of `std::vector<int>` and `std::string`.

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4  using namespace std;
5
6  int main() {
7      vector<int> v1;           // empty vector of ints
8      vector<int> v2(1000);     // vector containing 1000 zeros
9      vector<int> v3(1000, 8); // vector containing 1000 eights
10
11     string s1("drum");        // string containing 'd' 'r' 'u' and 'm'
12     string s2(8, 'a');        // string containing 8 'a's
13
14     return 0;
15 }
```

Line 9 is much nicer than having to write the following. Constructors make classes more convenient to use!

```
9      vector<int> v3;
10
11     for (size_t i = 0; i < 1000; ++i) {
12         v3.push_back(8);
13     }
```

In the previous examples, whenever we made an instance of `Rectangle`, we default constructed it, and then initialized the member variables `len1` and `len2`. For example, lines 7 to 9 of definition 15.1.3 said the following.

```
7      Rectangle r;
8      r.len1 = 2;
9      r.len2 = 4;
```

It would be much more convenient if we only had to write `Rectangle r(2, 4)`.

Definition 15.3.1. In the code below, lines 1 to 6 declare and define a struct called `Rectangle`.

```
1  struct Rectangle {
2      double len1;
3      double len2;
4
5      Rectangle(double _len1, double _len2) : len1(_len1), len2(_len2) {}
6  };
7
8  int main() {
9      Rectangle r(2, 4);
10
11     return 0;
12 }
```

- Line 5 declares and defines a *constructor* for the `Rectangle` struct. It looks a lot like a member function declaration and definition, but there are some important differences.
  - Its name is the same as the name of the struct. This is what tells you it is a constructor.
  - A return type is not specified. A constructor cannot return a value.
  - Between the closing parenthesis of the parameters and the opening brace of the body, it says `: len1(_len1), len2(_len2)`. This is called a *member initializer list* or a *constructor-initializer list*.

- To refer to the constructor on line 5 we use the name `Rectangle::Rectangle(double, double)`.

- The contents of the braces at the end of the definition are referred to as the *function body* just like for free functions and member functions.

- Line 9 uses `Rectangle::Rectangle(double, double)`.
  - This line constructs a new instance of `Rectangle` and names it `r`.
  - Due to the member initializer list, the member variables are given values `2.0` and `4.0`.

That definition introduced terminology, but the code execution still requires much explanation!

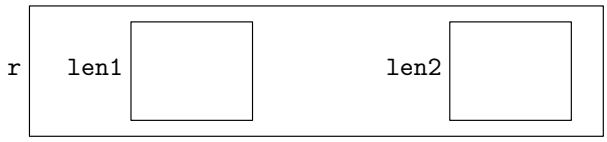
Example 15.3.2. Let's understand more carefully what happens when a constructor is used.

```
1   #include <iostream>
2   using namespace std;
3
4   struct Rectangle {
5       double len1;
6       double len2;
7
8       Rectangle(double _len1, double _len2) : len1(_len1), len2(_len2) {
9           if (len1 < 0 || len2 < 0) {
10              cout << "WARNING: the newly constructed instance of ";
11              cout << "Rectangle has a negative side length\n";
12          }
13      }
14  };
15
16  int main() {
17      Rectangle  r(2,  4);  cout <<  r.len1 << ' ' <<  r.len2 << "\n\n";
18      Rectangle sr(2, -4);  cout << sr.len1 << ' ' << sr.len2 << "\n\n";
19
20      return 0;
21  }
```

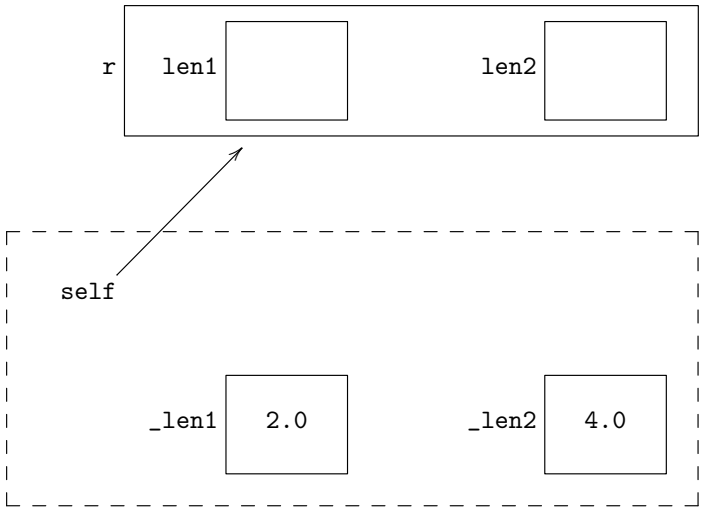
- Line 17 calls for an instance of `Rectangle` to be constructed using two `int`s.

  Just as for functions, constructors can be overloaded and a list of viable constructors will be considered by the compiler. Even though line 17 uses two `int`s, because an `int` can be cast to a `double`, `Rectangle::Rectangle(double, double)` is a viable constructor for resolving line 17. It is the only viable constructor and therefore the best viable constructor, so the compiler uses it.

- First, space is reserved in memory for a `Rectangle` called `r`. The order in which the member variables are declared in the struct interface determines the order in which they appear in memory.



- Then a scope is introduced.
  - The implicit parameter references the newly created instance of `Rectangle`.
  - The explicit arguments are assigned to the explicit parameters.
    In this case, the arguments `2` and `4` are cast to `2.0` and `4.0` and stored by the parameters.



- We are now ready to process the member initializer list:

  `: len1(_len1), len2(_len2).`

  Constructors frequently only do two things: create a new instance of a struct and initialize the newly created instance's member variables. The creation of a new instance is automatic when a constructor is used, as we have just seen. A member initializer list streamlines initializing the newly created instances' member variables. In our current situation...

  – `len1(_len1)` says

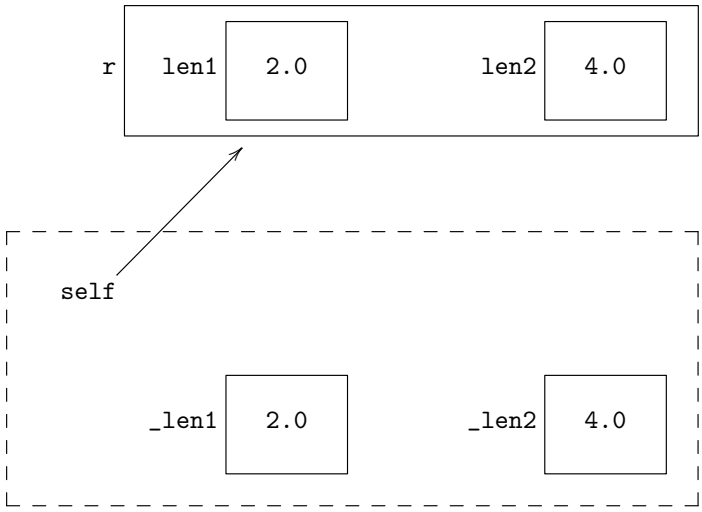
    "intitialize the member variable `len1` using the value of the parameter `_len1`".

    What member variable? Use the implicit parameter to make sense of it: `self.len1`.

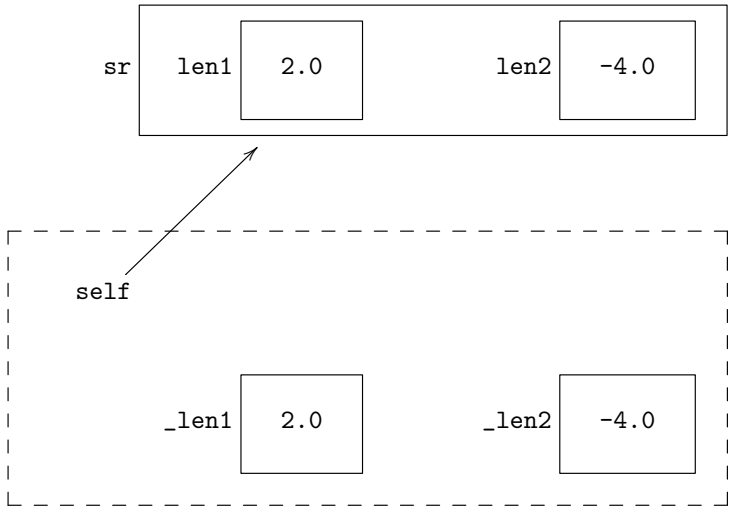
  – `len2(_len2)` says

    "intitialize the member variable `len2` using the value of the parameter `_len2`".

    What member variable? Use the implicit parameter to make sense of it: `self.len2`.



- Next, the function body executes. In this case, the condition within the `if` statement evaluates to `false` and so the function body does not accomplish much.
- Finally, the scope is destroyed, and we are left with `r`.
- Line 17 finishes by printing `2 4` and a couple of new lines.
- The execution of line 18 is really similar. `r` becomes `sr`, `4` becomes `-4`, and `4.0` becomes `-4.0`. However, this means that the condition in the `if` statement evaluates to `true` and so the function body causes an output saying "WARNING: the newly constructed instance of Rectangle has a negative side length".



- The final output is as follows.

```
2 4

WARNING: the newly constructed instance of Rectangle has a negative side length
2 -4
```

Definition 15.3.3. A constructor with zero parameters is called a *default constructor*.

A compiler-generated default constructor leaves member variables of fundamental type uninitialized and default constructs member variables of class type. A struct without any user-defined constructors is given a compiler-generated default constructor (as long as its definition would not create a build error). Once a struct has at least one user-defined constructor, the compiler will no longer consider generating a default constructor: if a user wants one, then they need to define one.

Example 15.3.4. The following code encounters a build error.

```
1   struct Rectangle {
2       double len1;
3       double len2;
4
5       Rectangle(double _len1, double _len2) : len1(_len1), len2(_len2) {}
6   };
7
8   int main() {
9       Rectangle r;
10      r.len1 = 2; r.len2 = 4;
11
12      return 0;
13  }
```

Line 9 asks to default construct an instance of `Rectangle`. Because the code includes a definition for `Rectangle::Rectangle(double, double)`, the compiler does not generate `Rectangle::Rectangle()`. `Rectangle::Rectangle(double, double)` is not viable because it has the incorrect number of parameters, so the build error is 14.1.10 (applied to constructors): there is no viable constructor and so there is no best viable constructor.

15.4 `const` instances and marking member functions `const`

Examples 15.2.2 and 15.2.6 left lingering questions about `const`. We had to delay answering these questions because a compiler-generated default constructor cannot be used to make a `const` instance of a struct that has a member variable of fundamental type (15.3.3 and 5.4.1). Now that we can write our own constructors, we can discuss `const` with examples. Everything we are about to discuss really follows from two facts.

Fact 1: When an instance of a struct is marked `const`, all of its member variables become `const`.
Fact 2: Later...

Example 15.4.1. The output of the code below is `2 4`.

Uncommenting line 16 or 17 would give build error 5.4.2. Since `r` is marked `const`, `r.len1` and `r.len2` both have type `const double`.

```
1  #include <iostream>
2  using namespace std;
3
4  struct Rectangle {
5      double len1;
6      double len2;
7
8      Rectangle(double _len1, double _len2) : len1(_len1), len2(_len2) {}
9  };
10
11 int main() {
12     const Rectangle r(2, 4);
13
14     cout << r.len1 << '␣' << r.len2 << endl;  // 2 4
15
16     // r.len1 = 3;  // build error since r.len1 is a 'const double'
17     // r.len2 = 6;  // build error since r.len2 is a 'const double'
18
19     return 0;
20 }
```

Example 15.4.2. This example shows that saying "const reference" instead of "reference to const" is incorrect. However, it is more advanced and can be safely ignored. Moreover, one should not take this example to imply that using member variables of reference type is very common.

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


struct C {
          string& r;
    const string& rc;

    C(string& _r, const string& _rc) : r(_r), rc(_rc) {}
};


int main() {
          string  s("non-const␣str");
    const string cs("␣␣␣␣const␣str");


    C a(s, cs);

    a.r  = "";
//  a.rc = "";          //     build error


    const C b(s, cs);

    b.r  = "";          // no build error!
//  b.rc = "";          //     build error


    return 0;
}
```

- `s` is a `std::string` and `cs` is a `const std::string`.
- `a` is an instance of `C`.

  Its member variable `r` is made to reference `s`.

  Its member variable `rc` is made to reference `cs`.
- Assigning to `a.r` is fine because its type is `std::string&`.
- Assigning to `a.rc` gives build error 10.5.2 because its type is `const std::string&`.
- `b` is a `const` instance of `C`. Like `a`, its member variable `r` is made to reference `s` and its member variable `rc` is made to reference `cs`.

- – Since `b` is marked `const`, "Fact 1" tells us that `b.r` is a `const` reference to a `std::string`.
  – If `b.r` was a reference to `const`, line 27 would give build error 10.5.2.
  – However, this line does *not* give a build error.

  This shows that when interpreting types like `const std::string&`, regarding the placement of `const` and what it means, there are some rules to learn. We will turn to some of these rules when we discuss pointers in section 17.

- Assigning to `b.rc` a gives build error just as for `a.rc`.

References to `const` and why references are already `const` will make more sense as you learn more!

Fact 1 makes sense: you should not be able to change a `const` instance of a struct, so you should not be able to change its member variables, so its member variables should be `const`.

To motivate Fact 2, recall...

- the implicit parameter is a reference;
- for explicit parameters of reference type we have to choose between a reference and a reference to `const`.

The next fact says that marking a member function `const` or not corresponds to saying whether the implicit parameter is a reference to `const` or a regular reference.

Fact 2: Suppose that `f` is a member function of a struct `C`.

- If `f` is not marked `const`, then the implicit parameter has type `C&`.
- If `f` is marked `const`, then the implicit parameter has type `const C&`, a reference to `const`.

Therefore...

a member function should be marked `const` when there is no way it can be responsible for mutating the instance that is referenced by the implicit parameter.

There are a number of consequences of Fact 1 and Fact 2.

Build Error 15.4.3. *It is a build error for a* `const` *instance of a struct to call a member function that is not marked* `const`*.*

Example 15.4.4. The following code encounters a build error due to line 8.

```
1   struct C {
2       void f() {}
3   };
4
5   int main() {
6       const C c;
7
8       c.f();
9
10      return 0;
11  }
```

The instance of `C` called `c` is `const`. On line 8, this instance `c` calls `f` which is not marked `const`.

A mathematician tries to remember as little as possible, and they may wish to note that this build error is a consequence of "Fact 2" and earlier build errors. Because of "Fact 2", the implicit-argument-to-implicit-parameter assignment for the function call on line 8 would be `C& self = c`. Since `c` is `const`, build error 10.5.3 means that `C::f` is not a viable candidate for the function call.

Example 15.4.5. The following code encounters a build error due to line 12.

```
1   struct Rectangle {
2       double len1, len2;
3
4       Rectangle(double _len1, double _len2) : len1(_len1), len2(_len2) {}
5
6       double area() { return len1 * len2; }
7   };
8
9   int main() {
10      const Rectangle r(2, 4);
11
12      r.area();
13
14      return 0;
15  }
```

The instance of `Rectangle` called `r` is `const`.
On line 12, this instance `r` calls `area` which is not marked `const`.

Build Error 15.4.6. *If a member function is marked* `const`*, within the function body it is a build error to assign to a (non-*`mutable`*) member variable of the implicit parameter of fundamental type. The same is normally true for member variables of class type too.*

Example 15.4.7. The following code encounters a build error due to line 5.

```
1   struct C {
2       int i;
3
4       void f() const {
5           i = 0;
6       }
7   };
8
9   int main() {
10      return 0;
11  }
```

`C::f` is marked `const`, but in the function body an assignment is made to a member variable of the implicit parameter `i`.

For the thinking-is-better-than-memorizing bunch, this build error follows from "Fact 1", "Fact 2", and how references to `const` work. "Fact 2" tells us the implicit-parameter of `C::f` has type `const C&`. This means that through the implicit parameter, the referenced instance of `C` is treated as though it is `const`. By "Fact 1", that means that through the implicit parameter, we cannot assign to a member variable of type `int` because it behaves like a `const int`.

Example 15.4.8. The following code encounters a build error due to line 5.

```
1   struct Rectangle {
2       double len1, len2;
3
4       double area() const {
5           len1 = 0;
6
7           return len1 * len2;
8       }
9   };
10
11  int main() {
12      return 0;
13  }
```

`Rectangle::area` is marked `const`, but in the function body an assignment is made to `len1`.

Example 15.4.9. The following code prints `12 24`.

The new part here is `quadruple_side_lengths` calling `double_side_lengths` through the implicit parameter on lines 15 and 16.

```
1  #include <iostream>
2  using namespace std;
3
4  struct Rectangle {
5      double len1, len2;
6
7      Rectangle(double _len1, double _len2) : len1(_len1), len2(_len2) {}
8
9      void double_side_lengths() {
10         len1 *= 2;
11         len2 *= 2;
12     }
13
14     void quadruple_side_lengths() {
15         double_side_lengths();
16         double_side_lengths();
17     }
18
19     double area() const {
20         return len1 * len2;
21     }
22 };
23
24 int main() {
25     Rectangle r(3, 6);
26
27     r.quadruple_side_lengths();
28
29     cout << r.len1 << '␣' << r.len2 << endl;
30
31     return 0;
32 }
```

This idea is introduced so that the next build error has some context. We will see other examples of this type of thing later on.

Build Error 15.4.10. *If a member function is marked* `const`*, within the function body it is a build error to call a member function which is not marked* `const` *through the implicit parameter.*

Example 15.4.11. The following code encounters a build error due to line 5.

```
1   struct C {
2       void nc() {}
3
4       void f() const {
5           nc();
6       }
7   };
8
9   int main() {
10      return 0;
11  }
```

`C::f` is marked `const` and calls `C::nc`, which is not marked `const`, through the implicit parameter.

This build error is a consequence of "Fact 2" and earlier build errors. Because of "Fact 2", the implicit-argument-to-implicit-parameter assignment for the function call on line 5 would be from `C::f`'s implicit parameter of type `const C&` to `C::nc`'s implicit parameter of type `C&`. Build error 10.5.3 means that `C::nc` is not a viable candidate for the function call.

Example 15.4.12. The following code encounters a build error due to line 10.

```
1   struct Rectangle {
2       double len1, len2;
3
4       void double_side_lengths() {
5           len1 *= 2;
6           len2 *= 2;
7       }
8
9       double area() const {
10          double_side_lengths();
11          return len1 * len2;
12      }
13  };
14
15  int main() {
16      return 0;
17  }
```

`Rectangle::area` is marked `const` and calls `Rectangle::double_side_lengths`, which is not marked `const`, through the implicit parameter.

15.5 Summary of marking member functions `const`

In the last three build-error-related `Rectangle` examples, we saw...

- When `area` is not marked `const`, we cannot ask for the area of a `const` `Rectangle`.
- When `area` is marked `const`, we are prevented from changing (the `len1` member variable of) the `Rectangle` that calls `area`.
- When `area` is marked `const`, we are prevented from calling `Rectangle::double_side_lengths()` through the implicit parameter because it is not marked `const`.

A member function should be marked `const` when there is no way it can be responsible for mutating instances that call it.

We can rethink the build errors above as follows...

- When a member function is not marked `const`, the compiler assumes it will change instances that call it. To protect us, the compiler does not build code that attempts to call this member function using a `const` instance.
- When a member function is marked `const`, we have told the compiler that the member function should not change instances that call it. If the code in our function body attempts to change the implicit parameter, to protect us, the compiler refuses to build.
- When a member function is marked `const`, we have told the compiler that the member function should *not* change instances that call it. Not marking another member function `const` leads to the compiler assuming that it *will* change the instances that call it. If we call that member function using the implicit parameter of the first member function, to protect us, the compiler refuses to build.

Example 15.5.1. Here is a summary example which builds and runs, but shows some build errors in the comments. `C::h` is defined outside of the struct interface as shown in 15.7.

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

struct C {
    int var;

    C() : var(0) {}

    void f()       { ++var; cout << "no␣const:␣" << var << endl; }
    void g() const {        cout << "␣␣␣const:␣" << var << endl; }
    void h() const;
};

int main() {
    C a;

    a.var = 1;

    a.f();        // Implicit-argument-to-implicit-parameter assignment is
                  // C& self = a;

    a.g();        // Implicit-argument-to-implicit-parameter assignment is
                  // const C& self = a;

    const C b;

//  b.var = 2;    // b is marked 'const' so its member variables becomes 'const'.

//  b.f();        // Implicit-argument-to-implicit-parameter assignment would be
                  // C& self = b;

                  // This would give a build error so 'C::f' is not viable.
                  // There is no viable function and a build error.

    b.g();        // Implicit-argument-to-implicit-parameter assignment is
                  // const C& self = b;

    return 0;
}

void C::h() const {
    // 'self' will have type 'const C&'.

    // var = 4;   // Build error.
    // f();       // Build error.

    g();
}
```

Example 15.5.2. The following more applied example also gives a useful summary.

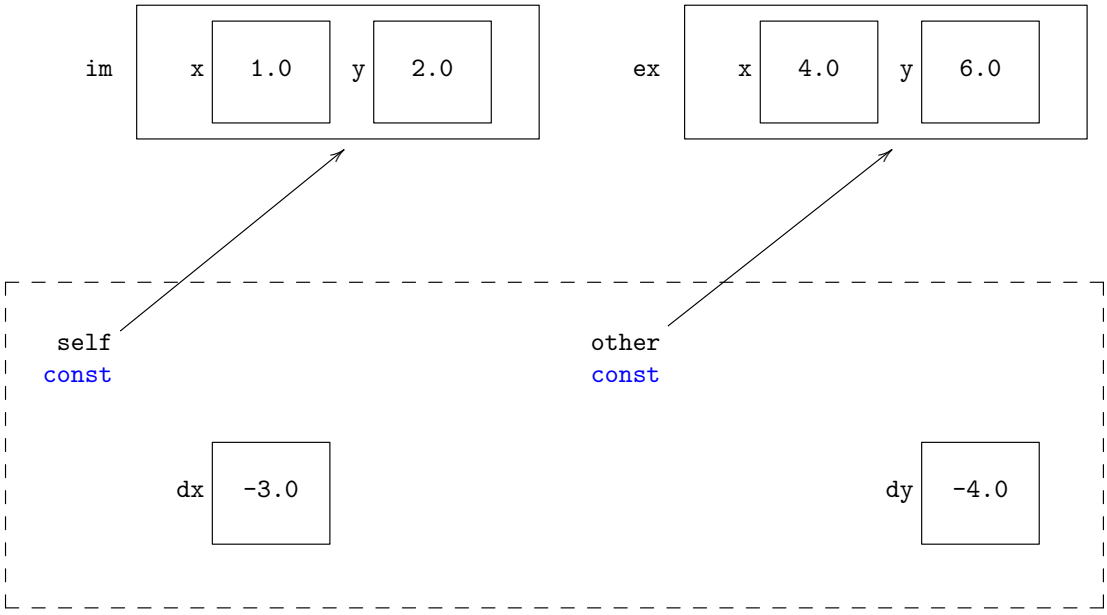
```
1   #include <iostream>
2   #include <cmath>
3   using namespace std;
4
5   struct Point {
6       double x;
7       double y;
8
9       Point(double _x, double _y) : x(_x), y(_y) {}
10
11      double length_to(const Point& other) const {
12          double  dx  =  x - other.x;
13          double  dy  =  y - other.y;
14
15          return  sqrt(dx * dx  +  dy * dy);
16      }
17  };
18
19  int main() {
20      Point im(1, 2);
21      Point ex(4, 6);
22
23      cout << im.length_to(ex) << endl;
24
25      return 0;
26  }
```

- Lines 5 to 17 define a struct called `Point`.
- Lines 6 and 7 tell us that instances of `Point` will have member variables called `x` and `y`.
- Line 9 declares and defines a constructor that one might refer to as the "obvious constructor":
  - its parameters correspond exactly to the member variables;
  - its member initializer list initializes each member variable using the corresponding parameter (disambiguated with an underscore);
  - its body does nothing.

  Line 20 and 21 use this constructor to construct instances of `Point` called `im` and `ex`.
- Lines 11 to 16 declare and define a member function called `Point::length_to`. On line 23 it is called by `im` and this line executes to print `5` because $5 = \sqrt{(1-4)\cdot(1-4)+(2-6)\cdot(2-6)}$.
- One line 23 `im` is the instance of `Point` that calls `Point::length_to`. It is the implicit argument to the function call. `ex` is the only explicit argument.

- When defining `length_to`, the explicit parameter is a reference to `const` because...
  - `Point` is a class type and there is a mild benefit in avoiding copying an instance of `Point`.
  - Asking for the length between two points does not change either point.
    In particular, calling `length_to` should not change the explicit argument.
- The implicit parameter is always a reference type. `length_to` is marked `const` to indicate that the implicit parameter is a reference to `const`. This is because...
  - Asking for the length between two points does not change either point.
    In particular, calling `length_to` should not change the implicit argument.

The picture for the function call on line 23 is drawn below.



Remember that during the execution of the code in function body, `x` and `y` will behave like `self.x` and `self.y`, accessing `1` and `4`. The fact that `self` and `other` are references to `const` means that they treat `im` and `ex` like `const Points`, and `self.x`, `self.y`, `other.x`, `other.y` behave like `const doubles`. Moreover, using references to `const` means that we can make `im` and `ex` `const Points` without producing a build error. This is good: we should be able to ask for the length between two `const Points`!

```
20      const Point im(1, 2);
21      const Point ex(4, 6);
22
23      cout << im.length_to(ex) << endl;
```

15.6 More on constructors

We have seen constructors using member initializer lists. We have used member initializer lists to initialize fundamental types (and references in one example), but we mentioned that they are also sometimes called *constructor*-initializer lists, and this name suggests that they should also be able to construct instances of classes. The next example demonstrates this.

Example 15.6.1. Below, lines 1 to 6 define a struct called `Point` with two member variables and the "obvious constructor" `Point::Point(double, double)`.

Lines 8 to 18 define a struct called `Triangle`. It has three member variables: `A`, `B`, and `C`. Each of these has type `Point`.

The `Point` interface serves as a blueprint for instances of `Point`. The `Triangle` interface serves as a blueprint for instances of `Triangle`, but this blueprint heavily relies on the previous one!

```
1  struct Point {
2      double x;
3      double y;
4
5      Point(double _x, double _y) : x(_x), y(_y) {}
6  };
7
8  struct Triangle {
9      Point A; // the vertices
10     Point B; // of the
11     Point C; // Triangle
12
13     Triangle(double A_x, double A_y,
14              double B_x, double B_y,
15              double C_x, double C_y) : A(A_x, A_y),
16                                        B(B_x, B_y),
17                                        C(C_x, C_y) {}
18 };
19
20 int main() {
21     Triangle t(1, 2,
22                3, 4,
23                5, 8);
24     return 0;
25 }
```

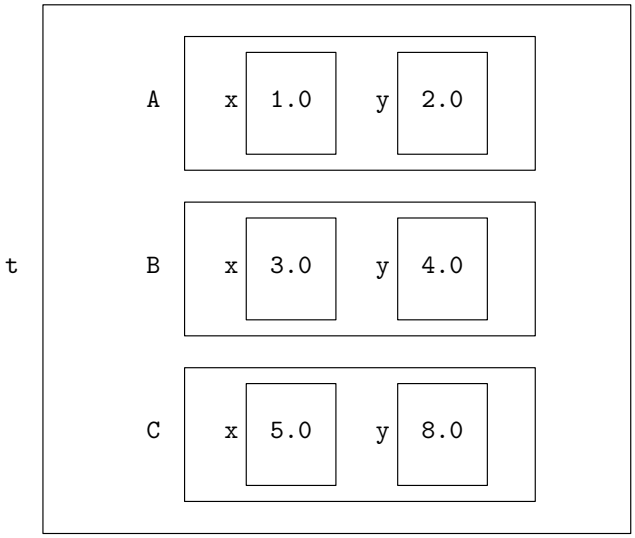
What about the `Triangle` constructor declared and defined on lines 13 to 17? It has six parameters of type `double`. That is the appropriate amount of information to specify three `Point`s each with two member variables of type `double`. We just have to understand the member initializer list.

- `A(A_x, A_y)` says to construct the member variable `A`, an instance of the `Point` struct, using the two `double` parameters `A_x` and `A_y` in that order.

- `B(B_x, B_y)` says to construct the member variable `B`, an instance of the `Point` struct, using the two `double` parameters `B_x` and `B_y` in that order.
- `C(C_x, C_y)` says to construct the member variable `C`, an instance of the `Point` struct, using the two `double` parameters `C_x` and `C_y` in that order.

There is a constructor that fits the three jobs perfectly: `Point::Point(double, double)`, the constructor declared and defined on line 5. On lines 21 to 23 when we create an instance of `Triangle`...

- `Triangle::Triangle(double, double, double, double, double, double)` is called once;
- `Point::Point(double, double)` is called three times.



This last example hints at the power of writing structs. As soon as you have defined one, you can define another using it. This is particularly powerful when you write useful member functions.

Example 15.6.2. To calculate the perimeter of a triangle, you need to know its side lengths. If you know the vertices of triangle, then you can calculate these lengths as the lengths between points, and so we can build on the previous two examples in an elegant way.

The triangle with vertices $(0,0)$, $(1,0)$, and $(0,1)$, has a perimeter of $1+1+\sqrt{2} = 2+\sqrt{2} \approx 3.41421$ and the following code builds and runs to produce an output of `3.41421`.

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

struct Point {
    double x;
    double y;

    Point(double _x, double _y) : x(_x), y(_y) {}

    double length_to(const Point& other) const {
        double  dx  =  x - other.x;
        double  dy  =  y - other.y;

        return  sqrt(dx * dx  +  dy * dy);
    }
};

struct Triangle {
    Point A;
    Point B;
    Point C;

    Triangle(double A_x, double A_y,
             double B_x, double B_y,
             double C_x, double C_y) : A(A_x, A_y),
                                       B(B_x, B_y),
                                       C(C_x, C_y) {}

    double perimeter() const {
        return A.length_to(B) + B.length_to(C) + C.length_to(A);
    }
};

int main() {
    Triangle t(0, 0,
               1, 0,
               0, 1);

    cout << t.perimeter() << endl;

    return 0;
}
```

Definition 15.6.3. Suppose that `C` is a struct.

A constructor with signature `C::C(const C&)` is called a *copy constructor* because it can be used to create an instance of `C` from a pre-existing instance of `C`.

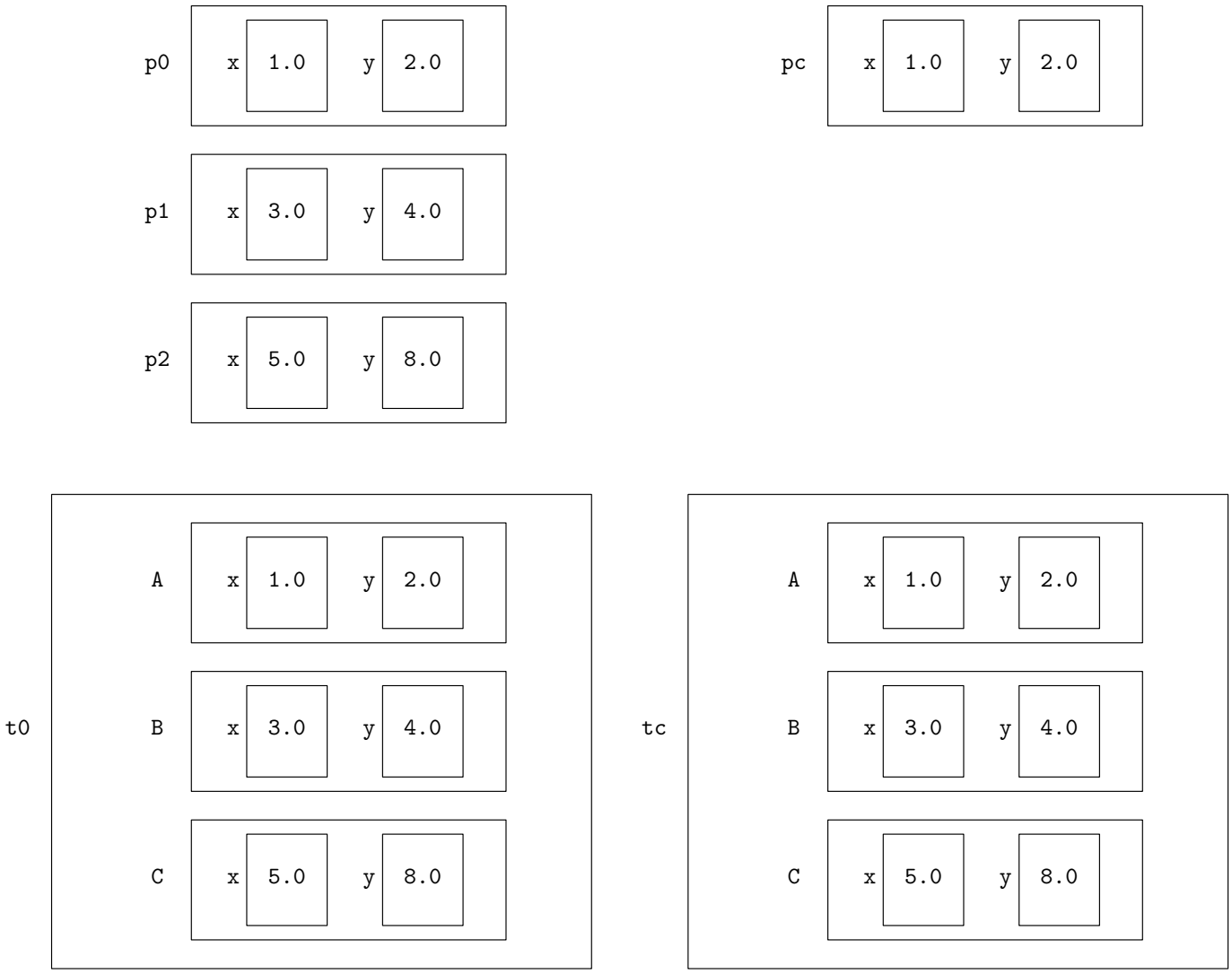
At this stage, when we are not thinking about defining our own copy (or move) constructors (or assignment operators, or destructors), we are always given a compiler-generated copy constructor. This constructor copies each member variable to make a new instance for us.

Example 15.6.4. One could wonder about the "obvious constructor" for `Triangle`. It has signature `Triangle::Triangle(const Point&, const Point&, const Point&)`. Each parameter has type `const Point&` corresponding to the fact that each member variable has type `Point`. Its member initializer list constructs each member variable using the corresponding parameter and its body does nothing. In this case `A(_A)` is asking to use the compiler-defined copy constructor which copies the two member variables `_A.x` and `_A.y` to `self.A.x` and `self.A.y`, respectively.

Lines 23 to 26 of the following code copy $6 + 2 + 6 = 14$ `double`s while executing.

```
1   struct Point {
2       double x;
3       double y;
4
5       Point(double _x, double _y) : x(_x), y(_y) {}
6   };
7
8   struct Triangle {
9       Point A;
10      Point B;
11      Point C;
12
13      Triangle(const Point& _A,
14               const Point& _B,
15               const Point& _C) : A(_A), B(_B), C(_C) {}
16  };
17
18  int main() {
19      Point p0(1, 2);
20      Point p1(3, 4);
21      Point p2(5, 8);
22
23      Triangle t0(p0, p1, p2);
24
25      Point    pc(p0);
26      Triangle tc(t0);
27
28      return 0;
29  }
```

- When `t0` is constructed on line 23, `p0`, `p1`, and `p2` are copied (using `Point`'s compiler-generated copy constructor) which involves copying `p0.x`, `p0.y`, `p1.x`, `p1.y`, `p2.x`, and `p2.y`.
- On line 25, `pc` is contructed by copying `p0` (using `Point`'s compiler-generated copy constructor) which involves copying `p0.x` and `p0.y`.
- `tc` is contructed by copying `t0` (using `Triangle`'s compiler-generated copy constructor) which involves copying `t0.A`, `t0.B`, and `t0.C` (using `Point`'s compiler-generated copy constructor) which involves copying `t0.A.x`, `t0.A.y`, `t0.B.x`, `t0.B.y`, `t0.C.x`, and `t0.C.y`.



Example 15.6.5. Overloading of member functions and constructors is allowed, and the resolution of overloads is similar for free functions, member functions, and constructors.

The following example constructs two instances of `Triangle`. The first is constructed from three instances of `Point` using the "obvious constructor". The second is constructed from six `int`s being cast to `double`s and `Triangle::Triangle(double, double, double, double, double, double)`.

```
struct Point {
    double x;
    double y;

    Point(double _x, double _y) : x(_x), y(_y) {}
};

struct Triangle {
    Point A;
    Point B;
    Point C;

    Triangle(const Point& _A,
             const Point& _B,
             const Point& _C) : A(_A), B(_B), C(_C) {}

    Triangle(double A_x, double A_y,
             double B_x, double B_y,
             double C_x, double C_y) : A(A_x, A_y),
                                       B(B_x, B_y),
                                       C(C_x, C_y) {}
};

int main() {
    Point p0(1, 2);
    Point p1(3, 4);
    Point p2(5, 8);

    Triangle t_from_3_Points(p0, p1, p2);

    Triangle t_from_6_doubles(0, 0,
                              1, 0,
                              0, 1);

    return 0;
}
```

Putting everything together, we now have two nice structs.

Example 15.6.6. The output of the following code is `3.41421`.

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

struct Point {
    double x;
    double y;

    Point(double _x, double _y) : x(_x), y(_y) {}

    double length_to(const Point& other) const {
        double  dx  =  x - other.x;
        double  dy  =  y - other.y;

        return  sqrt(dx * dx  +  dy * dy);
    }
};

struct Triangle {
    Point A;
    Point B;
    Point C;

    Triangle(const Point& _A,
             const Point& _B,
             const Point& _C) : A(_A), B(_B), C(_C) {}

    Triangle(double A_x, double A_y,
             double B_x, double B_y,
             double C_x, double C_y) : A(A_x, A_y),
                                       B(B_x, B_y),
                                       C(C_x, C_y) {}

    double perimeter() const {
        return A.length_to(B) + B.length_to(C) + C.length_to(A);
    }
};

int main() {
    cout << Triangle(0, 0,
                     1, 0,
                     0, 1).perimeter() << endl;
    return 0;
}
```

15.7 Defining member functions and constructors outside of the interface

Example 15.7.1. The output of the following code is `3.41421`.

The member functions are declared within the appropriate struct interface and defined outside the interface and after `main` with the use of the scoping operator `::` to specify the full name of the member function. This allows header and cpp separation.

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

struct Point {
    double x, y;

    Point(double _x, double _y);

    double length_to(const Point& other) const;
};

struct Triangle {
    Point A, B, C;

    Triangle(const Point& _A, const Point& _B, const Point& _C);

    double perimeter() const;
};

int main() {
    cout << Triangle(Point(0, 0),
                     Point(1, 0),
                     Point(0, 1)).perimeter() << endl;
    return 0;
}

double Triangle::perimeter() const {
    return A.length_to(B) + B.length_to(C) + C.length_to(A);
}

double Point::length_to(const Point& other) const {
    double  dx  =  x - other.x;
    double  dy  =  y - other.y;

    return  sqrt(dx * dx  +  dy * dy);
}

Triangle::Triangle(const Point& _A,
                   const Point& _B,
                   const Point& _C) : A(_A), B(_B), C(_C) {}

Point::Point(double _x, double _y) : x(_x), y(_y) {}
```

15.8 Header file and cpp arrangement

Suppose you are writing a struct and you wish to organize your code across a header and cpp file.

- The struct interface should occur within the header file
  (because it contains the member function declarations).
- Any member function or constructor definitions that are outside of the struct interface should be placed within the corresponding cpp file.

Example 15.8.1. Here we arrange our structs `Point` and `Triangle` across files called `Point.hpp`, `Point.cpp`, `Triangle.hpp`, and `Triangle.cpp`.

Point.hpp

```
#ifndef Point_hpp
#define Point_hpp


#include <cmath> // 'std::sqrt' needed for 'length_to'.


struct Point {
    double x, y;

    Point(double _x, double _y);

    double length_to(const Point& other) const;
};


#endif /* Point_hpp */
```

Point.cpp

```
#include "Point.hpp"


Point::Point(double _x, double _y) : x(_x), y(_y) {}


double Point::length_to(const Point& other) const {
    double  dx  =  x - other.x;
    double  dy  =  y - other.y;

    return  std::sqrt(dx * dx  +  dy * dy);
}
```

Triangle.hpp

```
#ifndef Triangle_hpp
#define Triangle_hpp


#include "Point.hpp" // 'Point' needed for member variables.


struct Triangle {
    Point A, B, C;

    Triangle(const Point& _A, const Point& _B, const Point& _C);

    Triangle(double A_x, double A_y,
             double B_x, double B_y,
             double C_x, double C_y);

    double perimeter() const;
};


#endif /* Triangle_hpp */
```

Triangle.cpp

```
#include "Triangle.hpp"


Triangle::Triangle(const Point& _A,
                   const Point& _B,
                   const Point& _C) : A(_A), B(_B), C(_C) {}


Triangle::Triangle(double A_x, double A_y,
                   double B_x, double B_y,
                   double C_x, double C_y) : A(A_x, A_y),
                                             B(B_x, B_y),
                                             C(C_x, C_y) {}


double Triangle::perimeter() const {
    return A.length_to(B) + B.length_to(C) + C.length_to(A);
}
```

main.cpp

```
1  #include <iostream>
2
3  #include "Point.hpp"
4  #include "Triangle.hpp"
5
6  using namespace std;
7
8
9  int main() {
10     const Point p0(0, 0);
11     const Point p1(3, 0);
12     const Point p2(3, 4);
13
14     cout << p0.length_to(p1) << '␣';
15     cout << p1.length_to(p2) << '␣';
16     cout << p2.length_to(p0) << '␣';
17
18
19     cout << Triangle(p0, p1, p2).perimeter() << endl;
20
21     cout << Triangle(0, 0,
22                      1, 0,
23                      0, 1).perimeter() << endl;
24
25     return 0;
26 }
```

Notice that `main.cpp` includes `Point.hpp` and `Triangle.hpp`.

- This is reasonable because the `main` function uses both the `Point` and the `Triangle` struct.
- Since `Triangle.hpp` includes `Point.hpp`, this means we have asked to include `Point.hpp` twice.
- `Point.hpp` includes a **definition** of the `Point` struct. Redefining a struct (in the same translation unit) gives a build error, but the *header guards* on lines 1, 2, and 17 that contain the code in `Point.hpp` prevent the code from being included twice, and so they save us from such a build error.

```
1  #ifndef Point_hpp
2  #define Point_hpp
16
17 #endif /* Point_hpp */
```

The code builds and runs to produce the following output.

```
1  3 4 5 12
2  3.41421
```

15.9 The `public` and `private` keywords and classes

Definition 15.9.1. Unless we specify otherwise, the members of a struct are `public`. This allows us to use the member access operator to access the members of instances (and to use constructors) without any restrictions. This has been useful for teaching and learning purposes.

A class is different to a struct in exactly one way: unless we specify otherwise, the members of a class are `private`. This means that we can only use the member access operator to access the members of an instance (or use constructors) from within the definitions of the class's member functions and constructors. Making all members of a class private makes the class unusable.

One can write the `public` and `private` keywords within the interface of a struct or class followed by a colon to specify the accessibility of the members which follow. Often public members are listed before private members.

Example 15.9.2. The following code allows for a thorough explanation of the previous definition.

```
#include <iostream>

class A {
public:
    A() : i(0) {}

    void  print_i() const { std::cout << i << '␣'; f(); }
    void mutate_i()       {             ++i;        f(); }

    void mutate_other_i(A& other) const;

private:
    int i;

    void f() const {}
};

void A::mutate_other_i(A& other) const { ++other.i; other.f(); }

int main() {
    A a0, a1;

//  a0.i;     // Build error.
//  a0.f();   // Build error.

    a0.print_i();
    a0.mutate_i();
    a0.print_i();

    a1.print_i();
    a0.mutate_other_i(a1);
    a1.print_i();

    return 0;
}
```

- Lines 3 to 16 define a class called `A`. The accessibility of all members of `A` are specified. Lines 5 to 10 list `A`'s public members. Lines 13 to 15 list `A`'s private members.
- Line 13 lists `A`'s only member variable. It is private and it is called `i`.
- Line 5 defines a public default constructor. It uses a member initializer list to initialize `i`. It is fine that `i` is used on this line before its listing on line 13.
- Line 21 default constructs two instances of `A` called `a0` and `a1`.

  If the constructor `A::A()` was private, line 21 would have created two build errors.
- Since `i` is private, uncommenting line 23 would create a build error. We are prevented from accessing `i` outside of the definitions of the class's member functions and constructors.
- On line 15 a private member function called `A::f` is declared and defined.
- Since `A::f` is private, uncommenting line 24 would create a build error. We are prevented from accessing `A::f` outside of the definitions of the class's member functions and constructors.
- On line 7 a public member function called `A::print_i` is declared and defined. In its definition it uses the private members `i` and `f` (before their listings on line 13 and 15). Access to `i` and `f` is allowed within the definitions of `A`'s member functions.
- On line 26 `A::print_i` is called by `a0`. This results in printing `a0.i` which is `0`. Although it is a build error to write `a0.i` within `main`'s definition, we can observe its value using this public member function.

  The member function call on line 26 also results in `a0.f()` being called even though it is a build error to write `a0.f()` within `main`'s definition. If you desire console evidence that `a0.f()` is called, you can edit `A::f`'s definition to include a useful print statement.
- On line 8 a public member function called `A::mutate_i` is declared and defined. In its definition it uses the private members `i` and `f`.
- On line 27 `A::mutate_i` is called by `a0`. This results in `a0.i`'s value being increased to `1`. This is highlighted by the member function call `a0.print_i()` on line 28.

  The member function call on line 27 also results in `a0.f()` being called again.
- Line 10 declares a member function called `A::mutate_other_i` and line 18 provides its definition. In its definition we see that it accesses the `i` member variable of an instance of `A` called `other`. This is allowed because we are within the definition of one of `A`'s member functions. We can also see that `other` calls `A::f` and this is allowed for the same reason.
- On line 31, `a0.mutate_other_i(a1)` causes `a1.i`'s value to be increased to `1` and for `a1.f()` to execute.
- The final output is `0 1 0 1 .`

Build Error 15.9.3. *It is a build error to access a private member of a struct or class outside of the definitions of the class's member functions and constructors.*

Summarizing, the only difference between structs and classes is the accessibility of members that are not explicitly listed as `public` or `private`. However, normally it is best to specify the accessibility of all members explicitly. Therefore, unless it is convenient for everything to be public and we use a struct, we will normally write a class, list all public members first followed by all private members.

Why we would ever want to prevent access to a member variable or member function? This is a little lengthy to describe...

Classes are designed to be used by people — for example, we have used the `std::string` class and the `std::vector` class template — and they often create the illusion of being much simpler than they are. Did you know that a `std::string` uses a `char*` to store the location of its first `char`? Probably not because we have not learned what a `char*` is yet. Importantly, this did not prevent you from using the `std::string` class. However, imagine if the first thing you found upon looking up the `std::string` was that it has a member variable called `ptr` of type `char*`. Upon creating an instance of `std::string` called `s`, you might consider assigning to `s.ptr` and this would completely destroy the functionality of `s`.

The summary: some members of a class are important for the correct working of the class, but a user of the class does not need to know about them, and moreover, allowing the user access to them could make the user's experience worse. When you look up `std::string`, it says nothing about a member variable of type `char*` called `ptr` because any similar member variable will have been made private. However, to be honest, you can get the value of the `char*` by calling `std::string::data`.

The utility of being able to hide members from a user becomes clearer and clearer as you write more classes that make use of memory management. This is a huge topic in PIC 10B. For now, we will consider a simpler example: fractions or *rational numbers*.

Example 15.9.4. The following class is written to store rational numbers. "Rational number" is just a fancy name for "fraction". A fraction $\frac{n}{d}$ consists of an integer numerator n and denominator d and there are rules of cancellation so that $\frac{3}{6} = \frac{1}{2}$. Therefore, it makes sense for us to give this class two member variables of type `int`: `num` and `den`.

The member variables are made private because a user might do strange things if they are given direct access to `num` and `den`. A user of the class will surely want to print, compare, add, subtract, multiply, and divide fractions, and so we give them public member functions for these operations. However, randomly adding 7 to a numerator of a fraction sounds like it may be a mistake, and not something we want to allow a user to do. We can give an even better reason for making `num` and `den` private though...

`int`s have a limited range and they encounter overflow when arithmetic is performed that results in values outside of this range. To ensure our `Rational` class can avoid overflow as much as possible, it makes sense to store our fractions in simplified form. We will ensure:

- `den` is strictly positive,
- `num` and `den` have a highest common factor of 1.

So we store the rational number $\frac{-2}{-16} = \frac{1}{8}$ by setting `num` to `1` and `den` to `8`. If a user was allowed to access `num`, and they increased its value by 7, we would have `num` set to `8` and `den` to `8`. We'd be storing $\frac{8}{8} = \frac{1}{1}$, but not in its simplified form. Some of our definitions make critical use of the fact that we store our fractions in simplified form. For example, `Rational::equals` compares two numerators and two denominators. This only gives the correct answer when the two fractions have been simplified: $\frac{3}{6}$ and $\frac{1}{2}$ do not have equal numerators or equal denominators. Because of all of this, it is really important we do not give users direct access to the `num` and `den` member variables.

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

class Rational {
public:
    Rational()             : num(0), den(1) {}
    Rational(int n)        : num(n), den(1) {}
    Rational(int n, int d) : num(n), den(d) {
        assert(den != 0);
        make_den_positive();
        simplify();
    }

    void print() const {
        cout << num << '/' << den;
    }

    bool equals(const Rational& other) const {
        return (num == other.num) && (den == other.den);
    }

    bool is_zero() const { return num == 0; }

    Rational       plus(const Rational& other) const;
    Rational      minus(const Rational& other) const;
    Rational      times(const Rational& other) const;
    Rational divided_by(const Rational& other) const;

private:
    int num;
    int den;

    void make_den_positive() {
        if (den < 0) { num *= -1; den *= -1; }
    }

    void simplify() {
        int a = abs(num), b = abs(den);

        while (b != 0) {
            int r = a % b;
            a = b;
            b = r;
        }

        num /= a; den /= a;
    }
};
```

Lines 6 to 50 define the `Rational` class. Its public members are listed from line 8 to line 29 and its private members are listed from line 32 to line 49.

Lines 8 to 14 provide three overloads for the constructor. The default constructor creates $0 = \frac{0}{1}$. Any integer can be regarded as a fraction with denominator equal to 1, and the second constructor makes use of this idea to construct $n = \frac{n}{1}$.

The third constructor allows a user to specify a numerator and denominator. `assert(den != 0)` on line 11 will cause a runtime error if the specified denominator is `0`. Calls to the private member functions defined on lines 35 to 49 (which a user of the class does not need to know about) make sure the rational number is stored in its simplified form. `Rational::make_den_positive` is simple. `Rational::simplify` uses the Euclidean algorithm which might be unfamiliar to you.

The definitions of `Rational::print`, `Rational::equals`, and `Rational::is_zero` on lines 16 to 24 are nice and short. These member functions are made public because they will be useful for a user to the class. For the same reason, the member functions declared on lines 26 to 29 are public. The definitions of `Rational::add`, `Rational::minus`, `Rational::times`, `Rational::divided_by` are provided below.

```
52  Rational Rational::plus(const Rational& other) const {
53      return Rational(num * other.den + other.num * den, den * other.den);
54  }
55  Rational Rational::minus(const Rational& other) const {
56      return Rational(num * other.den - other.num * den, den * other.den);
57  }
58  Rational Rational::times(const Rational& other) const {
59      return Rational(num * other.num, den * other.den);
60  }
61  Rational Rational::divided_by(const Rational& other) const {
62      assert(!other.is_zero());
63      return Rational(num * other.den, den * other.num);
64  }
```

Their definitions come from the mathematical formulae:

$$\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 \cdot d_2 + n_2 \cdot d_1}{d_1 \cdot d_2} \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 \cdot d_2 - n_2 \cdot d_1}{d_1 \cdot d_2} \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 \cdot n_2}{d_1 \cdot d_2} \\
\frac{n_1}{d_1} / \frac{n_2}{d_2} &= \frac{n_1 \cdot d_2}{d_1 \cdot n_2}
\end{aligned}$$

In each case we call `Rational::Rational(int, int)` directly. We can write `Rational(3, 6)` to construct the instance of `Rational` with `num` set to `1` and `den` set to `2` (recall that this constructor does all the necessary simplifying) and in the definitions above, we construct instances of `Rational` on the same line that they are returned.

Finally, in `main` we perform the following calculations:

$$2 + \frac{\frac{3}{4}}{\frac{1}{5}} = 2 + \frac{15}{4} = \frac{23}{4}, \qquad \frac{2 + \frac{3}{4}}{\frac{1}{5}} = \frac{\frac{11}{4}}{\frac{1}{5}} = \frac{55}{4}$$

```
66  int main() {
67      Rational(2).plus(Rational(3, 4).divided_by(Rational(1, 5))).print();
68      cout << endl;
69
70      Rational(2).plus(Rational(3, 4)).divided_by(Rational(1, 5)).print();
71      cout << endl;
72
73      return 0;
74  }
```

The output is as follows.

```
1   23/4
2   55/4
```

Remark 15.9.5. The lines in `main` above are quite horrible to read. It would be much nicer if they said the following.

```
66  int main() {
67      cout << Rational(2) + (Rational(3, 4) / Rational(1, 5));
68      cout << endl;
69
70      cout << (Rational(2) + Rational(3, 4)) / Rational(1, 5);
71      cout << endl;
72
73      return 0;
74  }
```

PIC 10B is the answer to your dreams because you need to overload `operator<<`, `operator+`, and `operator/` for these lines to execute as desired. Upon learning that content, you will also become aware of other deficiencies in the way the class above is written. However, for now, it provides a good example to become more familiar with classes.

15.10 Even more on constructors, advocating for member initializer lists

In this section, you will learn that you can investigate when constructors are called by giving them print statements. The examples will show you lots.

- Member variables are initialized and constructed in the same order that they are declared.
- If a member initializer list does not list a member variable of class type, that member variable is default constructed before the function body is reached.
- Member variables of a class type that does not have a default constructor must be listed in every member initializer list.
- If a member initializer list does not list a member variable of fundamental type, that member variable is left uninitialized.
- `const` member variables of fundamental type must be listed in every member initializer list.

Based on these rules, it makes most sense to...

- **always write a member initializer list that lists all of the member variables in the order that they are declared.**

These rules show that if a class `C` has a compiler-generated default constructor, it behaves just like the constructor with definition `C::C() {}`.

You will also see that for class types, copying often amounts to calling a copy constructor. In many other situations, however, it will amount to calling a copy assignment operator. There is lots of good stuff in PIC 10B!

Example 15.10.1. The code below builds and runs to produce the following output.

```
A0
A1
A2
A3
```

We see that the print statements in the constructors' definitions reveal exactly when each is used. Also, since the parameters are not used, we did not have to name them.

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

struct A {
    A()          { cout << "A0␣"; }
    A(int)       { cout << "A1␣"; }
    A(int, int)  { cout << "A2␣"; }
    A(const A&)  { cout << "A3␣"; }
};

int main() {
    A a0;          cout << endl;
    A a1(0);       cout << endl;
    A a2(0, 0);    cout << endl;
    A a3(a0);      cout << endl;

    return 0;
}
```

Example 15.10.2. The code below builds and runs to produce the following output.

```
A1 A2 B0
A1 A2 B1
A0 A0 B2 A2 A1
```

This highlights two things...

- Member variables are constructed in the order that they are declared. `B`'s member variable `x` is declared before the member variable `y`. Therefore, whenever an instance of `B` is constructed, the member variable `x` is constructed before `y`. This is highlighted by `A1` being printed before `A2` when `B::B()` and `B::B(int)` are used, even though in the second case `y` is written before `x` in the member initializer list.
- When we do not use a member initializer list, the member variables of class type are default constructed. This is highlighted by the two `A0`s being printed before `B2` when `B::B(double)` is used.

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

struct A {
    A()          { cout << "A0␣"; }
    A(int)       { cout << "A1␣"; }
    A(int, int)  { cout << "A2␣"; }
    A(const A&)  { cout << "A3␣"; }
};

struct B {
    A x; A y;

    B()        : x(0), y(0, 0) { cout << "B0␣"; }
    B(int)     : y(0, 0), x(0) { cout << "B1␣"; }
    B(double)                  { cout << "B2␣";  y = A(0, 0); x = A(0); }
};

int main() {
    B b0;          cout << endl;
    B b1(0);       cout << endl;
    B b2(0.0);     cout << endl;

    return 0;
}
```

Example 15.10.3. When a member variable of fundamental type is not initialized with the member initializer list, it is left uninitialized. The following code has undefined behavior. On line 14, `C::C()` is called to construct an instance of `C` called `c` and on line 8, it prints the uninitialized `c.j`. Recently, this code ran to print random-looking numbers on XCode and Visual Studio, and `0` on Online GDB.

Uncommenting line 10 would produce a build error because this constructor leaves the `const` member variable `i` uninitialized (5.4.1).

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

struct C {
    const int i;
          int j;

    C() : i(0)  { cout << j << endl; }

//  C(int) {}
};

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

Example 15.10.4. The code below builds and runs to produce the following output.

```
A0
A2 A3 B3
A3 A2 A3 B4
A3 A3
```

The `A3`s before `B3` and `B4` come from `y(a)` using `A`'s copy constructor. The `A3` at the start of the third line comes the argument-to-parameter assignment `A a = a0`. The two `A3`s on the last line show that the compiler-generated copy constructor for `B` copy constructs both of `B`'s member variables.

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

struct A {
    A()          { cout << "A0␣"; }
    A(int)       { cout << "A1␣"; }
    A(int, int) { cout << "A2␣"; }
    A(const A&) { cout << "A3␣"; }
};

struct B {
    A x; A y;

    B(A& a)       : x(0, 0), y(a) { cout << "B3␣"; }
    B(A  a, int) : x(0, 0), y(a) { cout << "B4␣"; }
};

int main() {
    A a0;          cout << endl;

    B b3(a0);      cout << endl;
    B b4(a0, 0);   cout << endl;
    B b5(b4);      cout << endl;

    return 0;
}
```

Example 15.10.5. This extended example contains the previous examples.

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

struct A {
    A()          { cout << "A0␣"; }
    A(int)       { cout << "A1␣"; }
    A(int, int) { cout << "A2␣"; }
    A(const A&) { cout << "A3␣"; }
};

struct B {
    A x; A y;

    B()           : x(0), y(0, 0) { cout << "B0␣"; }
    B(int)        : y(0, 0), x(0) { cout << "B1␣"; }
    B(double)                     { cout << "B2␣";  y = A(0, 0); x = A(0); }

    B(A& a)       : x(0, 0), y(a) { cout << "B3␣"; }
    B(A  a, int) : x(0, 0), y(a) { cout << "B4␣"; }
};

struct C {
    const int i;
          int j;

    C() : i(0)  { cout << j << endl; }

//  C(int) {}  // build error
};

int main() {
    A a0;          cout << endl;
    A a1(0);       cout << endl;
    A a2(0, 0);    cout << endl;
    A a3(a0);      cout << endl << endl;

    B b0;          cout << endl;
    B b1(0);       cout << endl;
    B b2(0.0);     cout << endl << endl;

    B b3(a0);      cout << endl;
    B b4(a0, 0);   cout << endl;
    B b5(b4);      cout << endl << endl;

//  C c;        // undefined behavior

    return 0;
}
```

The code builds and runs to produce the following output.

```
A0
A1
A2
A3

A1 A2 B0
A1 A2 B1
A0 A0 B2 A2 A1

A2 A3 B3
A3 A2 A3 B4
A3 A3
```

We saw in some of the previous examples that uncommenting line 28 would give a build error and that uncommenting line 45 would give undefined behavior. On the other hand, commenting line 5 would create three build errors. Since three other constructors for `A` are defined, `A` would not have a compiler-generated default constructor. Therefore, line 16 would give two build errors due to the empty member initializer list asking to default construct `x` and `y`, and line 32 would also give a build error.

15.11 Review questions

1. The following code builds and runs.

```
struct Example {
    int i;
    double d();
    char c;
    bool b();
    unsigned int u;
};

int main() {
    Example e;
    Example x, a, m;

    return 0;
}

double Example::d() { return 0.0;   }
bool   Example::b() { return false; }
```

   - How many member variables does `Example` have?
   - How many member functions does `Example` have?
   - How many instances of `Example` are constructed?

2. The following code builds and runs.

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

struct Rectangle {
    double len1;
    double len2;

    void double_side_lengths() {
        len1 *= 2;
        len2 *= 2;
    }
    double area()      const { return      len1 * len2;  }
    double perimeter() const { return 2 * (len1 + len2); }
};

```

```
int main() {
    Rectangle rec;
    rec.len1 = 3;
    rec.len2 = 5;

    rec.double_side_lengths();
    cout << rec.perimeter() << endl;

    return 0;
}
```

What is the output?

3. The following code builds and runs. What is the output?

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

struct Example {
    int i;
    double d;

    int test(char c, bool b) {
        if (b) {
            return c;
        }

        i = c;
        return d;
    }
};

int main() {
    Example e;
    e.i = 1;
    e.d = 2.3;

    Example f;
    f.i = 4;
    f.d = 5.6;

    cout << e.test(static_cast<char>(78), true) << endl;
    cout << e.i << '␣' << e.d << endl;

    cout << f.test(static_cast<char>(90), false) << endl;
    cout << f.i << '␣' << f.d << endl;

    return 0;
}
```

4. (a) The following code builds and runs. What is the output?

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

struct C {
    int    mv1;
    int    mv2;
    double mv3;

    C(int a, double b, double c) : mv1(a), mv2(b), mv3(c) {
        cout <<  a  << '␣' <<  b  << '␣' <<  c  << endl;
        cout << mv1 << '␣' << mv2 << '␣' << mv3 << endl;
    }
};

int main() {
    C c(1.1, 2.2, 3.3);
    cout << c.mv1 << '␣' << c.mv2 << '␣' << c.mv3 << endl;

    return 0;
}
```

(b) The following code builds and runs. What is the output?

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

struct C {
    int    mv1;
    int    mv2;
    double mv3;

    C(int a, double b, double c) : mv1(a), mv2(b), mv3(c) {
        cout <<  a  << '␣' <<  b  << '␣' <<  c  << endl;
        cout << mv1 << '␣' << mv2 << '␣' << mv3 << endl;
    }

    C(double a, double b, double c) : mv1(a), mv2(b), mv3(c) {}
};

int main() {
    C c(1.1, 2.2, 3.3);
    cout << c.mv1 << '␣' << c.mv2 << '␣' << c.mv3 << endl;

    return 0;
}
```

5. How many build errors does the following code produce?
   Which lines produce the build errors?

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

struct X {
    int i;

    X(int _i) : i(_i) {}

    void f() const {}
    void g()       {}

    void h() const { ++i; }
    void j()       { ++i; }

    void p() const { g(); }
    void q()       { g(); }
};

int main() {
          X  x(0);
    const X cx(0);

     x.f();
    cx.f();

     x.g();
    cx.g();

    return 0;
}
```

6. Consider the following code.

   - If it encounters a build error, try to explain the build error as well as you can.
   - If it builds and executes successfully on all compilers, write down the output.

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

struct X {
    void f() const { cout << '1' << endl; }

    void g()       { cout << '2' << endl; }
    void g() const { cout << '3' << endl; }
};
```

```
10
11  int main() {
12              X  x;
13      const X cx;
14
15       x.f();
16      cx.f();
17
18       x.g();
19      cx.g();
20
21      return 0;
22  }
```

7. Consider the following code.

   - If it encounters a build error, try to explain the build error as well as you can.
   - If it builds and executes successfully on all compilers, write down the output.

```
1   #include <iostream>
2   using namespace std;
3
4   struct C {
5       void f();
6   };
7
8   void f() {
9       cout << 'f' << endl;
10  }
11
12  int main() {
13      C c;
14      c.f();
15
16      return 0;
17  }
```

8. (a) Consider the following code.

- If it encounters a build error, try to explain the build error as well as you can.
- If it builds and executes successfully on all compilers, write down the output.

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




class A {
public:
    A()        {         cout << "A::A()" << endl; }

    void f()   {   g(); cout << "A::f()" << endl; }


private:
    void g()   {         cout << "A::g()" << endl; }
};




class B {
public:
    B() : a() {         cout << "B::B()" << endl; }

    void g()   {   f(); cout << "B::g()" << endl; }


private:
    void f()   { a.g(); cout << "B::f()" << endl; }

    A a;
};




int main() {
    B b; b.g();

    return 0;
}
```

(b) Consider the following code.

- If it encounters a build error, try to explain the build error as well as you can.
- If it builds and executes successfully on all compilers, write down the output.

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




class A {
public:
    A()        {         cout << "A::A()" << endl; }

    void f()   {   g(); cout << "A::f()" << endl; }


private:
    void g()   {         cout << "A::g()" << endl; }
};




class B {
public:
    B() : a() {         cout << "B::B()" << endl; }

    void g()   { a.f(); cout << "B::g()" << endl; }


private:
    void f()   {   g(); cout << "B::f()" << endl; }

    A a;
};




int main() {
    B b; b.g();

    return 0;
}
```

9. What is the output of the following code?

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

struct X {
    int j;

    X(double d1, double d2) : j(d1 + d2) {
        cout << "d1 + d2" << endl; print();
    }

    void print() const { cout << "j = " << j << endl; }

    X(double d) : j(d) { cout << "d = " << d << endl; print(); }
    X(int    i) : j(i) { cout << "i = " << i << endl; print(); }
    X(         ) : j(8) {                              print(); }
};

struct Y {
    X x1; X x2; X x3;

    void print() const { cout << "x1.j = " << x1.j << endl;
                         cout << "x2.j = " << x2.j << endl;
                         cout << "x3.j = " << x3.j << endl << endl; }

    Y(               )                              {          }
    Y(int i          ) : x1(i), x2(0), x3(0)        { print(); }
    Y(int i, double d) : x2(d), x1(i), x3(d, 0.8)   { print(); }
};

int main() {
    Y y(1, 2.4);
    Y z;

    return 0;
}
```

16 Default arguments

It can be useful to specify a value for a function parameter to fall back on for when a corresponding argument is not provided by a call to the function. Such a value is called a *default argument*.

Example 16.1. The code below builds and runs to produce the following output.

```
1  Hello, Michael!!!
2  Hello, other living entity!!!
```

When the function `greet` is defined, a default argument of `"other␣living␣entity"` is specified for the parameter called `name`.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  void greet(const string& name = "other␣living␣entity") {
6      cout << "Hello,␣" << name << "!!!" << endl;
7  }
8
9  int main() {
10     greet("Michael");
11     greet();
12
13     return 0;
14 }
```

- The first function call on line 10 does not use the default argument because an argument is provided. As we have seen previously, the function call amounts to...

```
10     {
11         const string& name = "Michael";
12         cout << "Hello,␣" << name << "!!!" << endl;
13     }
```

- The second function call on line 11 does not provide an argument. Because of this, the default argument is used. The function call amounts to...

```
11     {
12         const string& name = "other␣living␣entity";
13         cout << "Hello,␣" << name << "!!!" << endl;
14     }
```

Example 16.2. The code below builds and runs to produce the following output.

```
1 1 2 3
1 1 2 3 5 8 13 21
1 1 2 3 5 8 13 21
```

When the function `print` is defined, a default argument of `static_cast<size_t>(-1)` is specified for the parameter called `up_to`.

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

/**
 Prints the values contained within a vector of ints.

 If an index is passed as the second argument,
 the function prints the elements up to that index.

 If the specified index is
 bigger than or equal to the size of the vector,
 all values contained within the vector are printed.

 If no index is specified,
 all values contained within the vector are printed.

 @param v : the vector of ints to print values from
 @param up_to : will not print beyond this index
*/
void print(const vector<int>& v, size_t up_to = -1) {
    up_to = min(up_to, v.size());

    for (size_t i = 0; i < up_to; ++i) {
        cout << v[i] << ' ';
    }
    cout << endl;
}

int main() {
    vector<int> v { 1, 1, 2, 3, 5, 8, 13, 21 };

    print(v, 4);
    print(v, 12);
    print(v);

    return 0;
}
```

When the `print(v, 4)` executes, the second argument-to-parameter assignment is `size_t up_to = 4`. Since `v.size()` returns `8`, line 23 of the function body leaves `up_to` unchanged, and the first `4` values of the vector are printed.

When line 35 executes, the second argument-to-parameter assignment is `size_t up_to = 12`. Since `v.size()` returns `8`, line 23 of the function body updates `up_to` to store `8`, and all `8` values of the vector are printed.

When `print(v)` is called, a second argument is not supplied, so the default argument for the second parameter is used. The second argument-to-parameter assignment becomes `size_t up_to = -1`. This means that `up_to` stores a very large number (either $2^{32}-1$ or $2^{64}-1$). Since `v.size()` returns `8`, line 23 of the function body updates `up_to` to store `8`, and all `8` values of the vector are printed.

You may have thought that it would be simpler to define `print` as follows.

```
22  void print(const vector<int>& v, size_t up_to = v.size()) {
23      for (size_t i = 0; i < up_to; ++i) {
24          cout << v[i] << '␣';
25      }
26      cout << endl;
27  }
```

However, a default argument is not allowed to be specified in terms of another parameter, and so this attempt at a definition would give a build error.

Build Error 16.3. *Defining a default argument in terms of another parameter gives a build error.*

Example 16.4. The following code encounters a build error because the parameter `j` is given a default argument of `i` and `i` is the other parameter.

```
1   void f(int i, int j = i) {}
2
3   int main() {
4       return 0;
5   }
```

Build Error 16.5. *If a function's parameter is given a default argument, every parameter to the right of that parameter must also be given a default argument.*

Otherwise, there will be a build error.

Example 16.6. The following code encounters a build error because the parameter `i` is given a default argument of `0` and `j` is to the right of `i`, but it is not given a default argument.

```
1   void f(int i = 0, int j) {}
2
3   int main() {
4       return 0;
5   }
```

Example 16.7. If a function is declared and defined separately, a default argument should only be specified once. If the declaration and definition take place in header and cpp files, you are advised to specify the default argument in the header file. In the following code, the `print` member function of the `Point` class has a default argument and it is only specified on line 10 of `Point.hpp`. It is not specified again on line 4 of `Point.cpp`.

Point.hpp

```
1  #ifndef Point_hpp
2  #define Point_hpp
3
4  #include <iostream>
5
6  class Point {
7  public:
8      Point(double _x, double _y) : x(_x), y(_y) {}
9
10     void print(bool use_angles = false) const;
11
12 private:
13     double x, y;
14 };
15
16 #endif /* Point_hpp */
```

Point.cpp

```
1  #include "Point.hpp"
2  using namespace std;
3
4  void Point::print(bool use_angles) const {
5      if (use_angles) {
6          cout << '<' << x << ",␣" << y << '>' << endl;
7      }
8      else {
9          cout << '(' << x << ",␣" << y << ')' << endl;
10     }
11 }
```

main.cpp

```
1  #include "Point.hpp"
2
3  int main() {
4      Point p(1, 2);
5
6      p.print(false);   // (1, 2)
7      p.print(true);    // <1, 2>
8      p.print();        // (1, 2)
9
10     return 0;
11 }
```

The output produced is highlighted in the comments in `main.cpp`.

Build Error 16.8. *It is a build error to redefine a default argument.*

Example 16.9. Changing line 4 of `Point.cpp` in the previous example to say the following would give a build error.

```
4  void Point::print(bool use_angles = false) const {
```

Example 16.10. This example demonstrates a function with more than one default argument.

```
1  #include <iostream>
2  using namespace std;
3
4  void f(int i1,     int i2,
5         int d1 = 8, int d2 = 88, int d3 = 888) {
6      cout << i1 << '␣' << i2                        << endl;
7      cout << d1 << '␣' << d2 << '␣' << d3 << endl << endl;
8  }
9
10 int main() {
11     f(1, 11);              //     uses all three default arguments
12     f(2, 22, 5);           // uses the last two default arguments
13     f(3, 33, 6, 66);       //      uses the last default argument
14     f(4, 44, 7, 77, 777); //  does not use any default arguments
15
16     return 0;
17 }
```

The code builds and runs to produce the following output.

```
1 11
8 88 888

2 22
5 88 888

3 33
6 66 888

4 44
7 77 777
```

Some member functions that you have already seen use default arguments. Here are links to some relevant cplusplus.com pages: string::find, string::rfind, and string::npos. You can see that the member function `std::string::find` uses a default argument of `0` to start looking at the beginning of a `std::string` if no position is specified. You can see that the member function `std::string::rfind` uses a default argument of `static_cast<size_t>(-1)` to start looking at the end of a `std::string` if no position is specified. It is likely that this latter default argument works similarly to example 16.2.

17 Pointers

17.1 Memory addresses and the dereferencing operator

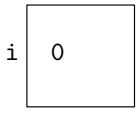
I have previously mentioned in passing that the `std::string` class and the `std::vector` class template make use of heap memory. In PIC 10B, you learn all about how these classes use pointers to perform their job. In PIC 10A, there is only an introduction to pointers. However, you will learn enough to explain how string literals can be passed to functions (without needing the `std::string` class). Considering that we first saw the string literal `"Hello, World!"` all the way back in section 3, it is good to finish PIC 10A by solving this problem.

Definition 17.1.1. Given any non-reference type `T`, there is another type `T*`. This type is read as "pointer to `T`". Pointer types allow us to store memory addresses.

Example 17.1.2. The first C++ type we learned a lot about was `int`, so let's begin with the type `int*`.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i = 0;
6
7      cout <<  i << endl; // 0
8      cout << &i << endl; // a memory address
9
10     int* p = &i;
11
12     cout <<  p << endl; // the same memory address
13     cout << *p << endl; // 0
14
15     *p = 1;
16
17     cout <<  i << endl; // 1
18     cout << *p << endl; // 1
19
20     return 0;
21 }
```

- We have seen line 5 many times. It tells the compiler to reserve enough space in memory for an `int`, that the name `i` will be used to access and modify the value stored at that place in memory, and to start by storing `0`.

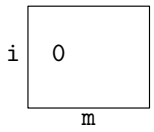


- Line 7 prints `i`'s value: `0`.
- Many locations on planet Earth have addresses. For example, "1 Oxford St, Cambridge, MA 02138" is the address of Harvard University. Because of this you should expect for locations in your computer to have memory addresses.

  On line 8 `&i` asks for the memory address of `i`.

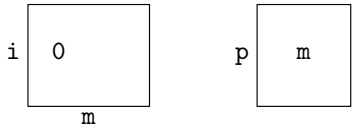
  - On XCode a memory address look like `0x16fdff288`.
  - On Online GDB one might be printed as `0x7fffc11a428c`.
  - On Visual Studio, they look a little different: `00000056628FF6C4`.

  Let's call the memory address `m` and draw it in the picture under the box that it addresses.



- `m` is the address of an `int`. The data type `int*` allows us to store it using a variable.

  On line 10 we declare a variable of type `int*` and initialize it using `m`.



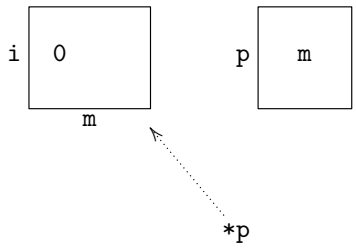
- On line 12 we print `p` which is printed however `&i` is printed, that is, however `m` is printed.
- On line 13 we write `*p`.

Definition 17.1.3. A pointer can be *dereferenced* using *the dereferencing operator* `*`.

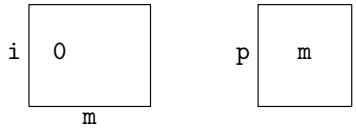
If `p` is a pointer storing an address `m`, `*p` provides the ability to access and modify what is at the location that `m` addresses.

Remark 17.1.4. A reference provides the ability to access and modify what is at a location. I do not wish to heavily advertise `*p` as a reference because definition 10.3.1 says a reference can only ever refer to *one* storage location and if you draw the reference as an arrow, the arrow is always going to start and end at the same place. `*p` is a function call to a function that returns by reference. It produces an arrow for some purpose and then the arrow goes away. (That purpose could be to initialize a more permanent reference.) The next time `*p` is called it may produce a different reference, that is, a different arrow. Since its existence is fleeting, I think it is best not to draw it, and to follow the address written in the box like a mail carrier.

If you forced me to draw `*p` in a picture, I would draw it like this, with the arrow perforated to emphasize its temporary nature.

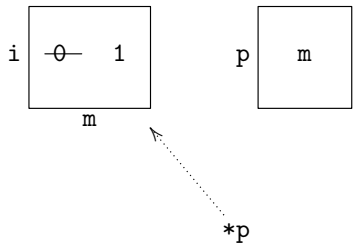


Line 13 executes to print `0` **and then the arrow is gone.**

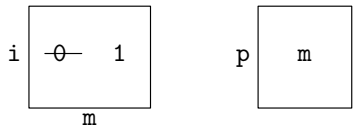


- Line 15 makes the assignment `*p = 1`.

  The arrow pops into existence to allow `0` to be overwritten with `1`...



  ...**and then the arrow is gone.**



- From this moment forward, I will never draw an arrow for a dereferenced pointer. Instead, **I will follow the address in the box.**

- Line 17 prints `i`'s value: `1`.

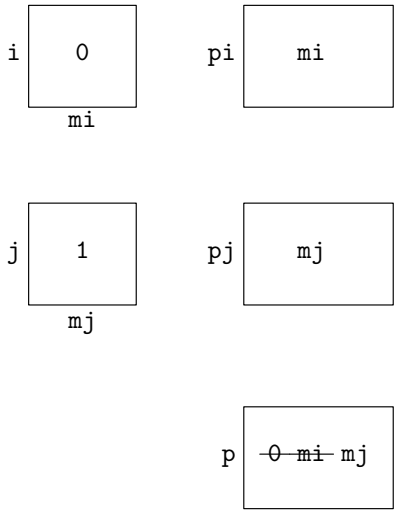
- Line 18 prints `*p`'s value which we obtain by looking in the box that is labelled by `p`, seeing the address `m`, and then going to that address where we find `1`.

Example 17.1.5. If you are not sure how to initialize a pointer, `nullptr` is often a good choice. In pictures, to abbreviate, I write `0`. `nullptr` creates the zero address, but writing `nullptr` is better than `0` because this way it cannot be regarded as an `int`.

Following the code below is a lot like chasing through other code involving variables. There are just a few points where we have to follow an address.

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

int main() {
    int   i =  0;
    int* pi = &i;

    int   j =  1;
    int* pj = &j;

    int* p  = nullptr;

    p = pi;

    cout <<  p << endl; // a memory address
    cout << *p << endl; // 0

    p = pj;

    cout <<  p << endl; // a different memory address
    cout << *p << endl; // 1

    return 0;
}
```

It is impossible to know the addresses of `i` and `j` until we run the code, so we just write `mi` and `mj`.



- When line 16 executes `p` stores `mi`, so we follow the address to `i` and `0` is printed.
- When line 21 executes `p` stores `mj`, so we follow the address to `j` and `1` is printed.

Remark 17.1.6. We have seen a couple of examples, but it is good to highlight the uses of syntax because both `&` and `*` are used in different ways. What they mean depends on whether they are used while specifying a type or not.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int i = 0;
6
7      // USED WITH A TYPE DURING DECLARATIONS
8      int& r = i;          // 'int&' is a type: "reference to an int".
9      int* p;              // 'int*' is a type: "pointer to an int".
10
11     p = &i;
12
13     // OTHERWISE
14     cout << &i << endl; // & is asking for the memory address of i.
15     cout << *p << endl; // * is dereferencing the pointer p.
16
17     return 0;
18 }
```

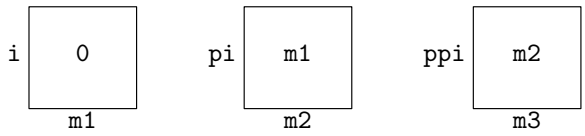
17.2 Pointers versus references, go... (Part 1)

In this section, we look at the biggest differences between references and pointers.

Example 17.2.1. The type `int**` is a pointer to a pointer to an `int`. It stores the memory address of a pointer to an `int`.

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

int main() {
    int     i =   0;
    int*   pi =  &i;
    int** ppi = &pi;

    cout << boolalpha;
    cout << ( *ppi == pi) << endl; // true: m1 == m1
    cout << (**ppi ==  i) << endl; // true:  0 == 0

    return 0;
}
```

In the code above, `i` has a memory address of `m1` which is stored by `pi` and `pi` has a memory address of `m2` which is stored by `ppi` (at an address of `m3`).



To understand dereferencing `ppi`...

- look at what is in the box labelled by `ppi`: `m2`;
- go to that address: you find `m1`.

Looking at the box labelled by `pi`, we also see `m1`, so `*ppi == pi` evaluates to `true`.

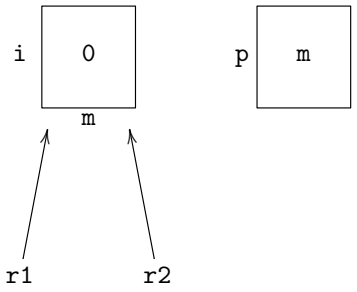
`**ppi` indicates to dereference twice: look in the box labelled by `ppi`, see `m2`, go there (`*`), see `m1`, go there (`*`), see `0`. Looking at the box labelled by `i`, we also see `0`, so `**ppi == i` evaluates to `true`.

Example 17.2.2. You have seen lines 5, 6, and 8 before.

Line 9 shows that a reference can be initialized by dereferencing a pointer. Because `p` stores `m`, `int& r2 = *p` makes `r2` reference what is at the memory address `m`.

Both outputs are `true`. They show that taking the memory address of a reference asks for the memory address of what is being referenced. Since both `r1` and `r2` reference `i`, taking the address of `r1` and `r2` is the same as taking the address of `i`.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int   i =  0;
6      int*  p = &i;
7
8      int& r1 =  i;
9      int& r2 = *p;
10
11     cout << boolalpha;
12     cout << (&r1 == &i) << endl; // true: m == m
13     cout << (&r2 == &i) << endl; // true: m == m
14
15     return 0;
16 }
```



Remark 17.2.3.

- Example 17.2.1 shows that you can obtain the address of a pointer and make use of a pointer to a pointer.
- On the other hand, example 17.2.2 shows that when you ask for the address of a reference, you end up getting the address of what is referenced.

We will return to these comments and provide greater detail after the next section.

17.3 Pointers and `const`

We saw references and then references to `const`.

Definition 17.3.1. Given some C++ type `T` that does not involve references or `const`, we have three other types: `const T*`, `T* const`, `const T* const`.

- `const T*` is read as "pointer to `const T`";
- `T* const` is read as "`const` pointer to `T`";
- `const T* const` is read as "`const` pointer to `const T`".

A `const` pointer can only store the memory address that it is initialized with.
If `p` is a pointer to `const`, then `*p` returns a reference to `const`.

Example 17.3.2. The following code builds, runs, and prints `5.5 3.3`.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      double d1 = 1.1;
6      double d2 = 2.2;
7
8            double*       _p_d = &d1;
9      const double*       _pcd = &d1;
10           double* const cp_d = &d1;
11     const double* const cpcd = &d1;
12
13     _p_d = &d2;
14     _pcd = &d2;
15 //  cp_d = &d2;   // b: cannot assign to a const pointer
16 //  cpcd = &d2;   // b: cannot assign to a const pointer
17
18     *_p_d = 3.3;
19 //  *_pcd = 4.4;  // b: cannot assign to a dereferenced pointer to const
20     *cp_d = 5.5;
21 //  *cpcd = 6.6;  // b: cannot assign to a dereferenced pointer to const
22
23     cout << d1 << '␣' << d2 << endl;  // 5.5 3.3
24
25     return 0;
26 }
```

In the code above...

- `_p_d` stands for "pointer to `double`";
- `_pcd` stands for "pointer to `const double`";
- `cp_d` stands for "`const` pointer to `double`";
- `cpcd` stands for "`const` pointer to `const double`".

Uncommenting any of lines 15, 16, 19, 21 produces a build error.

- The build errors on lines 15 and 16 are build error 5.4.2 but for pointers.
- The build errors on line 19 and 21 are build error 10.5.2 since dereferencing a pointer to `const` returns a reference to `const`.

Build Error 17.3.3. *Build errors 5.4.1 and 5.4.2 apply to `const` pointers as well.*

Build Error 17.3.4. *The address of a `const` variable cannot be assigned to a pointer that is not a pointer to `const`.*

Build Error 17.3.5. *A reference that is not a reference to `const` cannot be initialized by dereferencing a pointer to `const`.*

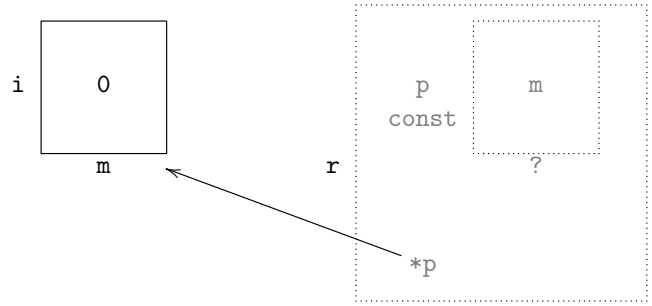
Example 17.3.6. The following code builds and runs. Uncommenting line 10 and 13 would create the build errors that were just described.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5            int     i  =    0;
6      const int    ci  =    0;
7
8      const int*  pc1  =   &i;
9      const int*  pc2  =  &ci;
10     //     int*   p   =  &ci;  // build error
11
12     const int&  rc   = *pc1;
13     //     int&   r   = *pc1;  // build error
14
15     return 0;
16 }
```

17.4 Pointers versus references, go... (Part 2)

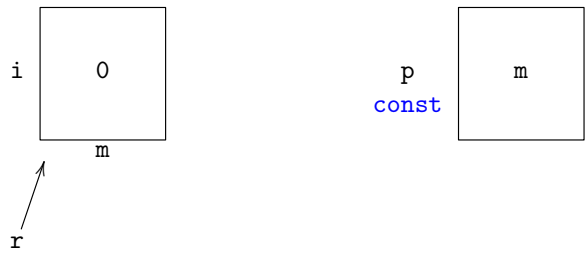
Remark 17.4.1. Under the hood, hidden from us, a reference is a `const` pointer, which is automatically dereferenced. It is for this reason that a reference must be initialized at its declaration (5.4.1 and 10.3.2) and we cannot change where the reference is pointing (10.3.1).

In example 17.2.1 we saw that you can obtain the address of a pointer and make use of a pointer to a pointer. However, for a reference, it is not possible to get the address of the underlying pointer. As we saw in example 17.2.2, because of the automatic dereferencing, we get the address of what is referenced, i.e. the address *stored by* the underlying pointer.



Example 17.4.2. Here is an example closely related to this discussion.

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

int main() {
    int         i =  0;
    int&        r =  i;
    int* const  p = &i;

    cout << boolalpha;

    cout <<    ( r == *p) << endl; // true: 0 == 0
    cout <<    (&r ==  p) << endl; // true: m == m

    cout << (*(&i) ==  i) << endl; // true: 0 == 0
    cout << (&(*p) ==  p) << endl; // true: m == m

    return 0;
}
```



17.5 Some things that are useful for PIC 10B: `this`, `->`, dereferencing `nullptr`

The content of this section can be safely ignored in PIC 10A.

Definition 17.5.1. Using `this` within a member function gives the memory address of the implicit parameter. Wherever `self` has been drawn earlier in this document, you can draw `*this`.

Example 17.5.2. The following code builds and runs to print `true true`.

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

struct C {
    C* get_address() {
        C& self = *this;
        return this;
    }
    const C* get_address() const {
        const C& self = *this;
        return this;
    }
};

int main() {
          C  c;
    const C cc;

    cout << boolalpha;
    cout << ( c.get_address() == &c ) << ' ';
    cout << (cc.get_address() == &cc) << endl;

    return 0;
}
```

Definition 17.5.3. Suppose that `C` is a class with a member function `C::f`. Suppose that `p` has type `C*`. Then `p->f(args...)` serves as a useful abbreviation for `(*p).f(args...)`.

Example 17.5.4. The following code builds and runs to print `13 13 13`.

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

int main() {
    string  s("I␣<3␣pointers");
    string* p = &s;

    cout <<    s.length() << '␣';
    cout << (*p).length() << '␣';
    cout <<   p->length() << endl;

    return 0;
}
```

Example 17.5.5. The following code builds and runs to print `000fff`.

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

class C {
public:
    C() : i(0) {}

    void example() {
        cout <<         i;
        cout << (*this).i;
        cout <<   this->i;

                 f();
        (*this).f();
          this->f();
    }

private:
    int i;

    void f() { cout << 'f'; }
};

int main() {
    C c;
    c.example();

    cout << endl;
    return 0;
}
```

Undefined Behavior 17.5.6. *Dereferencing a pointer that stores* `nullptr` *gives undefined behavior.*

Example 17.5.7. The following code produces undefined behavior.

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

int main() {
    int* p = nullptr;
    cout << *p << endl;

    return 0;
}
```

17.6 Review questions

1. What is the output produced by the following code?

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

int main() {
    string s1("1");
    string s2("2");

    string* p1 = &s1;
    string* p2 = &s2;

    swap(p1, p2);

    cout << s1 << s2 << *p1 << *p2 << endl;
    return 0;
}
```

2. What is the output produced by the following code?

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

int* give_address(int i) { return &i; }

int main() {
    int  i  =  1;
    int  j  = 11;

    int* p0 = &i;
    int* p1 = &i;
    int* p2 = &j;
    int* p3 = &j;

    int& r = *p0;

    *p0 += 1;

    cout <<  i  << "␣" <<  j                                  << endl;
    cout << *p0 << "␣" << *p1 << "␣" << *p2 << "␣" << *p3 << endl;

    p0 = p2;
    p2 = p1;

    *p0 *= 2;

    cout <<  i  << "␣" <<  j                                  << endl;
    cout << *p0 << "␣" << *p1 << "␣" << *p2 << "␣" << *p3 << endl;

    cout << boolalpha;
    cout << (&p1 == &p2) << "␣" << (give_address(i) == &i) << endl;

    cout << r << endl;

    return 0;
}
```

18 C-style Arrays

18.1 Introducing C-style arrays

"C-style array" refers to the notion of array inherited from the C language, what came before C++. C-style arrays feel like a less good `std::vector`. We mainly introduce them to help our discussion of pointer arithmetic.

Example 18.1.1. This example demonstrates three ways to declare and define a C-style array, and it shows that we can access a C-style array's elements using `[]` (just like for a `std::vector`).

```
1  #include <iostream>
2  #include <cstddef>
3  using namespace std;
4
5  int main() {
6      int arr_i[3];                       // { ???, ???, ??? }
7
8      char arr_c[] = { '4', '5' };        // { '4', '5' }
9
10     double arr_d[6] = { 7.7, 8.8 };     // { 7.7, 8.8, 0.0, 0.0, 0.0, 0.0 }
11
12
13     for (size_t idx = 0; idx < 3; ++idx) {
14         arr_i[idx] = 9;
15     }
16
17     for (size_t idx = 0; idx < 3; ++idx) {
18         cout << arr_i[idx] << '␣';
19     }
20     cout << endl;
21
22
23     for (size_t idx = 0; idx < 2; ++idx) {
24         cout << arr_c[idx] << '␣';
25     }
26     cout << endl;
27
28
29     for (size_t idx = 0; idx < 6; ++idx) {
30         cout << arr_d[idx] << '␣';
31     }
32     cout << endl;
33
34
35     return 0;
36 }
```

- Line 6 says that `arr_i` is an array of `int`s of size `3`. Its values are left uninitialized.

- Line 8 says that `arr_c` is an array of `char`s of size `2`. This size is deduced by the compiler using what is written on the right hand side of the assignment. The values are as specified: the `char`s `'4'` and `'5'`.
- Line 10 says that `arr_d` is an array of `double`s of size `6` and it specifies the first two values as `7.7` and `8.8`. The last four values are initialized to `0.0`.
- Lines 13 to 15 use a `for` loop to initialize `arr_i`'s values.
- Lines 17 to 32 use `for` loops to print all the elements. The output is as follows.

```
1    9 9 9
2    4 5
3    7.7 8.8 0 0 0 0
```

Here is something a little awkward about C-style arrays.

Undefined Behavior 18.1.2. *Specifying the size of a C-style array using an integer valued variable that is not `const` produces undefined behavior.*

Example 18.1.3. The following code encounters undefined behavior.

```
1    #include <cstddef>
2    using namespace std;
3
4    int main() {
5        size_t sz = 1;
6        int a[sz];
7
8        return 0;
9    }
```

Example 18.1.4. The following code does not encounter undefined behavior.

```
1    #include <cstddef>
2    using namespace std;
3
4    int main() {
5        const size_t sz = 1;
6        int a[sz];
7
8        return 0;
9    }
```

18.2 Pointer arithmetic

Definition 18.2.1. The `sizeof` operator can be used...

1. to ask how many bytes a data type uses;
2. to ask how many bytes the data type of an expression uses.

Example 18.2.2. The following code builds and runs.

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

int main() {
    cout << sizeof(char)     << '␣'; //      always evaluates to 1
    cout << sizeof(int)      << '␣'; // frequently evaluates to 4
    cout << sizeof(double)   << '␣'; // frequently evaluates to 8
    cout << sizeof(char[2]) << endl; //      always evaluates to 2

    cout << sizeof    '0'    << '␣'; //      always evaluates to 1
    cout << sizeof     0     << '␣'; // frequently evaluates to 4
    cout << sizeof    0.0    << '␣'; // frequently evaluates to 8
    cout << sizeof('0' + 0) << endl; // frequently evaluates to 4

    return 0;
}
```

The `char` datatype always uses 1 byte, the `int` datatype often uses 4 bytes, and the `double` datatype often uses 8 bytes. Therefore, the output of the code above is as follows for many compilers.

```
1 4 8 2
1 4 8 4
```

Definition 18.2.3. Suppose that `p` has type `T*` for some datatype `T` and `j` is a `size_t`. If `p` stores the address `m`, then `p + j` stores the address

```
m  +  j * sizeof(T)
```

provided that `p + j` does not give undefined behavior. This cppreference page details when `p + j` does not give undefined behavior and it amounts to when `m` and `m + j * sizeof(T)` are the addresses of elements in a suitable container or just past the last element of the container (sensible situations).

Definition 18.2.4. When expressing integers using the *decimal system*, 83526 means

$$6 \cdot 10^0 \ + \ 2 \cdot 10^1 \ + \ 5 \cdot 10^2 \ + \ 3 \cdot 10^3 \ + \ 8 \cdot 10^4.$$

The *binary* system replaces the role of 10 by 2. To distinguish decimal and binary representations, it can be useful to use the subscripts 10 and 2. To express 88_{10}, the number eighty-eight, in binary, we note that...

$$88 \ = \ 0 \cdot 2^0 \ + \ 0 \cdot 2^1 \ + \ 0 \cdot 2^2 \ + \ 1 \cdot 2^3 \ + \ 1 \cdot 2^4 \ + \ 0 \cdot 2^5 \ + \ 1 \cdot 2^6.$$

Therefore $88_{10} = 1011000_2$.

When counting in binary, we use two bits instead of ten digits and so numbers grow in length quicker. Counting from 0_{10} to 16_{10} becomes: 0_2, 1_2, 10_2, 11_2, 100_2, 101_2, 110_2, 111_2, 1000_2, 1001_2, 1010_2, 1011_2, 1100_2, 1101_2, 1110_2, 1111_2, 10000_2.

The *hexadecimal* system replaces the role of 10 by 16. We need more than ten digits and so we use the hexadecimal digits A, B, C, D, E, and F: $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, $C_{16} = 12_{10}$, $D_{16} = 13_{10}$, $E_{16} = 14_{10}$, $F_{16} = 15_{10}$.

$$\begin{aligned} 88 &= 8 \cdot 16^0 + 5 \cdot 16^1 \\ 127 &= 15 \cdot 16^0 + 7 \cdot 16^1 \\ 202 &= 10 \cdot 16^0 + 12 \cdot 16^1 \\ 255 &= 15 \cdot 16^0 + 15 \cdot 16^1 \end{aligned}$$

Therefore $88_{10} = 58_{16}$, $127_{10} = 7F_{16}$, $202_{10} = CA_{16}$, $255_{10} = FF_{16}$. When counting in hexadecimal, numbers grow in length slower. We got all the way up to 255_{10} only using two hexadecimal digits.

Example 18.2.5. Based on what cppreference says, the following code produces undefined behavior. However, at the time of writing, it builds and runs on XCode, Visual Studio, and Online GDB, and I would guess most compilers. Moreover, the output from building with those compilers nicely illustrates the previous three definitions.

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

int main() {
    cout << "sizeof(int)␣==␣" << sizeof(int) << "\n\n";

    int* p = nullptr;

    for (size_t j = 0; j <= 128; ++j) {
        cout << p + j << '␣';

        if (j % 4 == 3) {
            cout << endl;
        }
    }
    cout << endl;

    return 0;
}
```

The first output tells us that an `int` uses four bytes of memory. The code initializes `p` with `nullptr` and the `0`-th iteration of the `for` loop prints `p`. On XCode, `p`'s value is displayed as `0x0`. The next iteration of the `for` loop prints `p + 1` which is displayed as `0x4`. Adding `1` to `p` adds four bytes to the memory address it stores because `p` is an `int*` and an `int` uses four bytes of memory. Adding four to the memory address again to calculate `p + 2` gives `0x8`. Remember that the hexadecimal `C` stands for 12, that is $C_{16} = 12_{10}$, so adding four again gives `0xc`. Because $10_{16} = 16_{10}$, after moving onto a new line, we see `0x10`. You can then continue to see that pattern `0`, `4`, `8`, `c`, again and again as the leading hexadecimal goes up one by one. We see the multiples of four (from `0x0` to `0x200`) because an `int` uses four bytes of memory.

```
sizeof(int) == 4

0x0 0x4 0x8 0xc
0x10 0x14 0x18 0x1c
0x20 0x24 0x28 0x2c
0x30 0x34 0x38 0x3c
0x40 0x44 0x48 0x4c
0x50 0x54 0x58 0x5c
0x60 0x64 0x68 0x6c
0x70 0x74 0x78 0x7c
0x80 0x84 0x88 0x8c
0x90 0x94 0x98 0x9c
0xa0 0xa4 0xa8 0xac
0xb0 0xb4 0xb8 0xbc
0xc0 0xc4 0xc8 0xcc
0xd0 0xd4 0xd8 0xdc
0xe0 0xe4 0xe8 0xec
0xf0 0xf4 0xf8 0xfc
0x100 0x104 0x108 0x10c
0x110 0x114 0x118 0x11c
0x120 0x124 0x128 0x12c
0x130 0x134 0x138 0x13c
0x140 0x144 0x148 0x14c
0x150 0x154 0x158 0x15c
0x160 0x164 0x168 0x16c
0x170 0x174 0x178 0x17c
0x180 0x184 0x188 0x18c
0x190 0x194 0x198 0x19c
0x1a0 0x1a4 0x1a8 0x1ac
0x1b0 0x1b4 0x1b8 0x1bc
0x1c0 0x1c4 0x1c8 0x1cc
0x1d0 0x1d4 0x1d8 0x1dc
0x1e0 0x1e4 0x1e8 0x1ec
0x1f0 0x1f4 0x1f8 0x1fc
0x200
```

Example 18.2.6. This example is very similar to the previous one, but uses `double` instead of `int` and fewer iterations of the `for` loop.

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

int main() {
    cout << "sizeof(double) == " << sizeof(double) << "\n\n";

    double* p = nullptr;

    for (size_t j = 0; j <= 16; ++j) {
        cout << p + j << ' ';

        if (j % 2 == 1) {
            cout << endl;
        }
    }
    cout << endl;

    return 0;
}
```

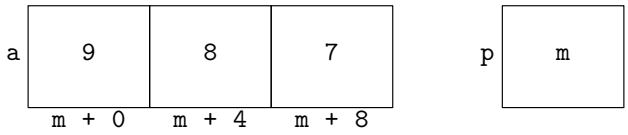
The first output tells us that a `double` uses eight bytes of memory, and so, because `p` is a `double*`, adding `1` to `p` adds eight bytes to the memory address that it stores. This time, the pattern is half as long — `0` and `8` again and again — because a `double` is twice as big as an `int` in memory. We see the multiples of eight (from `0x0` to `0x80`) because a `double` uses eight bytes of memory.

```
sizeof(double) == 8

0x0 0x8
0x10 0x18
0x20 0x28
0x30 0x38
0x40 0x48
0x50 0x58
0x60 0x68
0x70 0x78
0x80
```

Definition 18.2.7. A container that can be indexed using `[]` is said to store its data *contiguously* if...

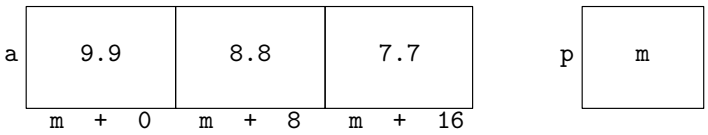
- `&a[0] + j == &a[j]` for all valid indices `j` of the container.

Example 18.2.8. C-style arrays store their data contiguously. Therefore, the following code prints `true` three times.




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

int main() {
    int a[] = { 9, 8, 7 };

    int* p = &a[0];

    cout << boolalpha;
    cout << (p + 0 == &a[0]) << endl; // true: m + 0 == m + 0
    cout << (p + 1 == &a[1]) << endl; // true: m + 4 == m + 4
    cout << (p + 2 == &a[2]) << endl; // true: m + 8 == m + 8

    return 0;
}
```

Example 18.2.9. This is almost the same example, but with `doubles`.




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

int main() {
    double a[] = { 9.9, 8.8, 7.7 };

    double* p = &a[0];

    cout << boolalpha;
    cout << (p + 0 == &a[0]) << endl; // true: m + 0  == m + 0
    cout << (p + 1 == &a[1]) << endl; // true: m + 8  == m + 8
    cout << (p + 2 == &a[2]) << endl; // true: m + 16 == m + 16

    return 0;
}
```

Definition 18.2.10. Suppose that `p` is a pointer and that `j` is a `size_t`. Then `p[j]` can be written as an abbreviation for `*(p + j)`.

Example 18.2.11. In the following code, each time that an expression like `*(p + j)` is used, `p[j]` behaves exactly the same way; and we access the `int`s in the C-style array one by one.

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

int main() {
    int a[] = { 9, 8, 7 };

    int* p = &a[0];

    cout << *(p + 0) << '␣' << p[0] << endl; // 9 9
    cout << *(p + 1) << '␣' << p[1] << endl; // 8 8
    cout << *(p + 2) << '␣' << p[2] << endl; // 7 7

    return 0;
}
```

Example 18.2.12. In this example, we access the `double`s in the C-style array one by one using expressions like `*(p + j)` and `p[j]`.

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

int main() {
    double a[] = { 9.9, 8.8, 7.7 };

    double* p = &a[0];

    cout << *(p + 0) << '␣' << p[0] << endl; // 9.9 9.9
    cout << *(p + 1) << '␣' << p[1] << endl; // 8.8 8.8
    cout << *(p + 2) << '␣' << p[2] << endl; // 7.7 7.7

    return 0;
}
```

We have seen that `sizeof(T)` affects arithmetic that uses pointers of type `T*`: the arithmetic for `int*`s and `double*`s behaves differently because `sizeof(int)` is not the same as `sizeof(double)`.

If `T1` and `T2` are distinct types, then the dereferencing operator behaves differently for pointers of type `T1*` and pointers of type `T2*`. The dereferencing operator interpets the bits and bytes differently, and as encoding different values. This allows us to have fun and games by taking a `double*`, casting it to a `int*`, and dereferencing it as two `int`s. The computer game Quake III moved between 32-bit integers and 32-bit floating points by casting between their pointers and dereferencing. This was to enable a fast inverse square root!

Example 18.2.13. `static_cast` is too well behaved to allow casting straight from a `double*` to an `int*`. However, it will allow casting from a `double*` to a `void*` to an `int*`. Its documentation for casting from `void*` explains that what we are doing here will give undefined behavior. That is because different architectures may encode `int` and `double` differently. However, XCode, Visual Studio, and Online GDB all build the following code and print `true true`.

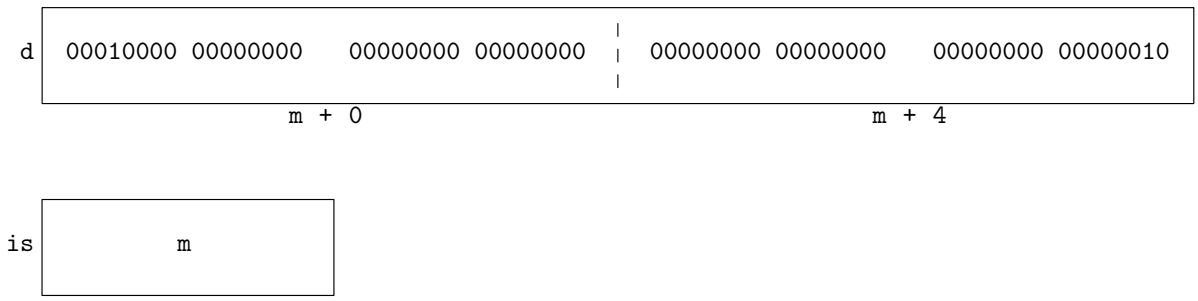
```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    cout << boolalpha;

    double d  = 2 + pow(2, -48);
    int*   is = static_cast<int*>(static_cast<void*>(&d));

    cout << (is[0] == 8)           << '␣'; // true
    cout << (is[1] == pow(2, 30)) << endl; // true

    return 0;
}
```

Recall definition 5.7.5.1. To represent $2 + 2^{-48}$ as a `double` we write it as $2(1 + 2^{-49})$ so that $s = 0$, $E = 1$, and $F = 2^{-49}$. In the order that the bits occur...

- $f_0 = f_1 = f_2 = 0$, $f_3 = 1$, and $f_i = 0$ for $3 < i < 52$.
- $e_i = 0$ for $i < 10$ and $e_{10} = 1$.
- $s = 0$.

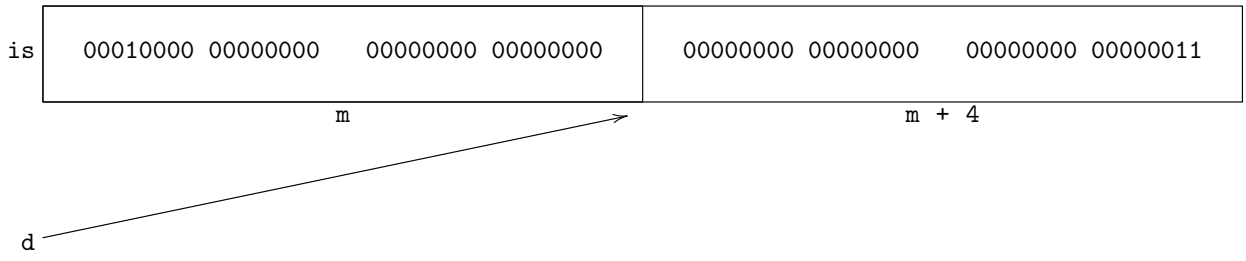
So (counting from 0) the 3-th and 62-th bit are 1 and the others are 0.

- $2^3 = 8$ and the 3-th bit is why `is[0]` is `8`.
- Since `int`s have 32 bits, the 62-th bit of `d` influences the $(62 - 32)$-th bit of `is[1]`. This is why `is[1]` is `static_cast<int>(pow(2, 30))`.



Example 18.2.14. In terms of the bits, this example is the same as example 18.2.13 except that the 63-th bit is also equal to 1. The code still encounters undefined behavior, but builds and runs to produce the same output on XCode, Visual Studio, and Online GDB, printing `true`. The code is arranged differently, highlighted most clearly by the picture and that line 9 *decays* `is` to a pointer, our next topic. (In the last two examples, it could be considered more accurate to write the bits in each byte in the opposite order, but I would not dwell on such ordering issues right now.)

```
1   #include <iostream>
2   #include <cmath>
3   using namespace std;
4
5   int main() {
6       cout << boolalpha;
7
8       int is[2];
9       double& d = *static_cast<double*>(static_cast<void*>(is));
10
11      is[0] = 8;
12      is[1] = -pow(2, 30);
13
14      cout << (-d == 2 + pow(2, -48)) << endl; // true
15
16      return 0;
17  }
```



18.3 Decaying C-style arrays to pointers and passing them to functions

Definition 18.3.1. Suppose that `arr` is a C-style array containing elements of type `T`. Then we can assign `arr` to a pointer of type `T*`. This behaves exactly like we assigned `&arr[0]`, that is, the memory address of the `0`-th element of the array.

When a C-style array has been replaced by the memory address of its `0`-th element, we say that the array has been *decayed* to a pointer.

Example 18.3.2. In the following code, on line 6, the array `a` is decayed to a pointer, and this pointer is stored using `p`.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a[] = { 9, 8, 7, 6, 5, 4 };
6      int* p = a;
7
8      cout << boolalpha;
9      cout << (p == &a[0]) << '␣';          // true
10
11     cout << sizeof a     << '␣';          // 24 (likely)
12     cout << sizeof p     << endl;         // 8  (likely)
13
14     return 0;
15 }
```

The output of the code is `true 24 8` on most machines. The `true` provides evidence that `p` stores the memory address of the `0`-th element of `a`. Because the array `a` contains 6 `int`s and each `int` uses 4 bytes (on most machines), the output of `24` follows from the multiplication $6 \cdot 4 = 24$.

On the other hand, `p` is not an array, it is a pointer, and pointers normally use 8 bytes of memory. After an array has been decayed to a pointer, the `sizeof` operator is not useful for deducing how much data the original array stored.

Remark 18.3.3. For a C-style array `a` and `size_t j`, `a[j]` can be interpreted in two ways...

- Access the `j`-th element of the array `a`.
- As an abbreviation for `*(a + j)`.

  In this expression, the pointer arithmetic `a + j` forces the array `a` to be decayed to a pointer.

Since C-style arrays store their data contiguously, the two interpretations both say to give a reference to the `j`-th element of the array `a`.

Example 18.3.4. Function parameter types that look like arrays are actually just pointers. Therefore, when passing an array to a function, we normally write a function where the parameter corresponding to the array is a pointer. This explicitly highlights that we are making use of decay. When writing such functions, we have to remember that the `sizeof` operator is not useful for determining information about an array that was passed to the function. This often necessitates passing in the size of an array as an additional argument.

In the following code, `increment` and `print` have parameters called `p` of type `int*` and `const int*`, respectively. In their function bodies, when we write `p[i]`, this is abbreviation for `*(p + i)`.

```
1   #include <iostream>
2   using namespace std;
3
4   void increment(int* p, size_t N) {
5       for (size_t i = 0; i < N; ++i) {
6           p[i] += 1;
7       }
8   }
9
10  void print(const int* p, size_t N) {
11      for (size_t i = 0; i < N; ++i) {
12          cout << p[i] << ' ';
13      }
14      cout << endl;
15  }
16
17  int main() {
18      int a[] = { 9, 8, 7, 6, 5, 4 };
19
20      print(a, 6);                              // 9 8 7 6 5 4
21      increment(a + 2, 4);
22      print(a, 6);                              // 9 8 8 7 6 5
23
24      return 0;
25  }
```

All the function calls make use of decay. For lines 20 and 22, the argument-to-parameter assignments are both `const int* p = a`; this code behaves like `const int* p = &a[0]`. For line 21, the argument-to-parameter assignment is `int* p = a + 2` which behaves like `int* p = &a[0] + 2`. The output is as follows.

```
1   9 8 7 6 5 4
2   9 8 8 7 6 5
```

Example 18.3.5. You may find it a little ridiculous that it is so difficult to print an array. This is because C-style arrays were inherited from C. If you wish to avoid passing in the size of an array to print, you need to use function templates and references to arrays. Function templates are discussed in PIC 10C and the syntax for a reference to an array has never made anybody happy.

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

template<size_t N>
void increment(int (&a)[N]) {
    for (size_t i = 0; i < N; ++i) {
        a[i] += 1;
    }
}

template<size_t N>
void print(const int (&a)[N]) {
    for (size_t i = 0; i < N; ++i) {
        cout << a[i] << '␣';
    }
    cout << endl;
}

int main() {
    int a1[] = { 1, 2, 3 };
    int a2[] = { 5, 6, 7, 8, 9 };

    print(a1);
    increment(a1);
    print(a1);

    print(a2);

    return 0;
}
```

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

18.4 String literals

Definition 18.4.1.

- The type of a *string literal* is an array of `const char`s. String literals are always one character longer than they look because the null character `'\0'` is added after the last specified character.
- A *C-string* is an array of `const char`s which includes `'\0'` exactly once as its last element.

Example 18.4.2. In the following example, uncommenting the first line gives a build error. `"Nope"` has type `const char[5]` because its five characters are `'N'`, `'o'`, `'p'`, `'e'`, and `'\0'`. We successfully store `"Test"` using a variable `c_str` of type `const char[5]` and show that its last character is `'\0'`.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5   //  const char _nope[4] = "Nope";     // "Nope" is too long for the array
6
7       const char c_str[5] = "Test";
8
9       cout << boolalpha;
10      cout << (c_str[4] == '\0') << endl;  // true
11
12      return 0;
13  }
```

Example 18.4.3. In this example we make use of decay from a `const char[8]` to `const char*` and from a `const char[17]` to `const char*`. The first decay occurs when we initialize `dc_str` on line 2, and the second occurs when we reassign `dc_str` on line 5. These reassignments are allowed because `dc_str` is **not** a `const` pointer. However, it **is** a pointer to `const`, which is why uncommenting line 4 would give a build error.

```
1   int main() {
2       const char* dc_str = "Testing";
3
4   //  dc_str[0] = 't';                // *dc_str is a reference to const
5       dc_str = "Testing, 1, 2...";
6
7       return 0;
8   }
```

Example 18.4.4. We are now able to define a function for printing C-strings. This function has the argument-to-parameter assignment perform decay to a `const char*`, uses `++` on this pointer to keep moving to the next character, and uses the null character to indicate when to stop printing.

The output of the following code is `Printing`.

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

void print(const char* s) {
    while (*s != '\0') {
        cout << *s;
        ++s;
    }
    cout << endl;
}

int main() {
    print("Printing");
    return 0;
}
```

Example 18.4.5. Observant readers might have noticed that the `print` function of the previous example fails to print a string literal that contains `'\0'` somewhere in the middle. Without using function templates or passing the length of the string literal, we cannot do better. Moreover, this is good enough for `cout <<` because C-strings, by definition, are terminated by the null character.

The output of the following code is `before null`. If you want to see a memory address when you print a `const char*`, you need to apply `static_cast<const void*>` first.

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

int main() {
    const char* p = "before␣null\0␣after␣null";
    cout << p << endl;

    return 0;
}
```

Example 18.4.6. The following example draws your attention to other "functions" that you have already used which pass string literals by decaying them to pointers `const char*`.

Line 6 uses the constructor `string::string(const char* s)`.

Lines 8, 9, and 10 use `size_t string::find(const char* s, size_t pos = 0) const`.

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

int main() {
    string s("mathematics");

    cout << s.find("math")    << '␣';  // 0
    cout << s.find("the")     << '␣';  // 2
    cout << s.find("mat", 2) << endl; // 5

    return 0;
}
```

18.5 A look towards PIC 10B

Example 18.5.1. The following code builds and runs and is incredibly likely to print `false`.

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

int main() {
    vector<int> v(4, 8);

    int* p_orig = &v[0]; v.reserve(2048);
    int* p_curr = &v[0];

    cout << boolalpha;
    cout << (p_orig == p_curr) << endl;

    return 0;
}
```

This shows that calling `v.reserve(2048)` on a `vector<int>` that was constructed with size `4` is very likely to cause the location of the `0`-th `int` to change in memory. Writing classes that can control such memory allocations is a major topic in PIC 10B.

18.6 Review questions

1. The following code builds and runs. What is its output?

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

int main() {
    int a[] = { 9, 6, 3, 0, 1, 2, 3, 4 };

    int* p = &a[0];

    p = p + 3;

    for (size_t j = 0; j < 3; ++j) {
        cout << p[2 * j] << '␣';
    }
    cout << endl;

    return 0;
}
```

2. The following code builds and runs. What is its output?

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

int main() {
    int  x[] = { 2, 4, 6, 8 };
    int  y = 7;
    int* z = &y;

    *z *= *(x + 1);
    *(x + 2) += *z;

    cout << x[0] << '␣' << x[1] << '␣' << x[2] << '␣' << x[3] << endl;
    cout << y << endl;

    return 0;
}
```

3. *Much more difficult.* You should use your IDE for this.

   Does the following code build and run?
   If so, what is its output and can you make any sense of it?

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

int main() {
    int i =       97
              +   98 * 256
              +   99 * 256 * 256
              +  100 * 256 * 256 * 256;

    int* pi = &i;

    char* pc = static_cast<char*>(static_cast<void*>(pi));

    cout << pc[0] << pc[1] << pc[2] << pc[3] << endl;

    return 0;
}
```