

Conversion Functions

The standard library has several functions, in the `<string>` header, that will convert a C++ `string` to a number (much like the C++11 `to_string()` function will convert a number to a `string`.)

- the `stoi(string)` function returns an `int`
- the `stod(string)` function returns a `double`
- There are also `stof()`, `stol()`, `stoul()` and `stold()` for the types `float`, `long`, `unsigned long` and `long double` respectively.

These functions **don't appear** in C++ 98. If your employer is using an older version of C++, and doesn't want to change to a later version, for a wide (and reasonable) variety of reasons. What can you do? Implement them yourself, of course.

This is not a made-up scenario; we are learning about the latest version of C++ here in CS 150, but in the real world, you'll need to be prepared for older versions.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Using the Conversion Functions

Your goal is to be able to compile **exactly the same file** in C++98, C++11, C++14 and C++17, and to **get the same results each time**. Click the "running-man" icon on the right to find our test program in **Replit**. Click the **Fork Repl** button so you have your own copy.



Switch to **Shell** window on the right and type **make 17**. You'll see that the program compiles and runs (although it terminates with an error when trying to convert "UB-40".)

```
>_ Console x Shell x +
~/Conversions$ make 17
g++ -std=c++17 -Wall -Werror -Wextra -pedantic -o conversions conversions.cpp
./conversions
stoi("42")->42
stod("3.14159")->3.14159
stoi("3.14159")->3
stod("4NonBlondes")->4
terminate called after throwing an instance of 'std::invalid_argument'
what(): stoi
make: *** [makefile:12: 17] Aborted (core dumped)
~/Conversions$
```

Now, type **make 98** in the Shell. This will compile your program with **C++98**.

```
>_ Console x Shell x +
~/Conversions$ make 98
g++ -std=c++98 -Wall -Werror -Wextra -pedantic -o conversions conversions.cpp
conversions.cpp: In function 'int main()':
conversions.cpp:7:33: error: 'stoi' was not declared in this scope
7 |     cout << "stoi(\"42\")->" << stoi("42") << endl;
  |                               ^~~~~
conversions.cpp:8:38: error: 'stod' was not declared in this scope; did you mean
'std'?
8 |     cout << "stod(\"3.14159\")->" << stod("3.14159") << endl;
  |                               ^~~~~
  |                               std
make: *** [makefile:15: 98] Error 1
~/Conversions$
```

OOPS! That doesn't look encouraging! What's wrong???

C++98 does not have those functions, so you get an **undeclared identifier**. You can fix that by **implementing these two functions** yourself, using the string stream classes.



This course content is offered under a [CC Attribution Non-Commercial](https://creativecommons.org/licenses/by-nc/4.0/) license. Content in this course can be considered under this license unless otherwise noted.

Stub the Replacements

Place these definitions for the functions right above `main`.

```
int stoi(const string& str) { return 0; }
double stod(const string& str) { return 0.0; }
```

Now, type `make 98` once again. You'll see that the code compiles and "runs" (although your stubs don't produce the correct value, of course).

What happens when you upgrade to Visual Studio 19? Will the code still compile? **Nope!** Your version of `stoi()` **conflicts with** the one already defined inside the new C++ standard library; we get a **clash of symbols**.



```
>_ Console x Shell x +
~/Conversions$ make 17
g++ -std=c++17 -o conversions conversions.cpp
conversions.cpp: In function 'int main()':
conversions.cpp:10:42: error: call of overloaded 'stoi(const char [3])' is ambiguous
  10 |         cout << "stoi(\"42\")->" << stoi("42") << endl;
      |                                     ^
conversions.cpp:5:5: note: candidate: 'int stoi(const string&)'
   5 | int stoi(const string& str) { return 0; }
```

At link time, there can be **only one copy** of the `stoi()` function in the executable; if the library already has one, **your program won't link**. This is called the **ODR** or **One Definition Rule**.

What you would like to say is: *"if I'm using C++11 or later use the library version, and, if I'm using an older version of C++, then use the version which I've written"*. You can do that with **conditional compilation**.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Using #define

One capability of the preprocessor is to substitute one portion of text for another, before your code is compiled. You do this with the preprocessor directive **#define**. Here's an example:

```
#define PI 3.14159265358979323846
```

This is called a **#defined** constant. Note that you do **not** use an equals sign when creating these constants. By convention, constants and functions (called **MACROs**) created by the preprocessor are named using all-caps.

With **PI** previously defined, if you write this fragment:

```
cout << PI << endl;
```

The preprocessor **replaces** the symbol **PI** with its predefined value before sending it to the compiler, which only sees this:

```
cout << 3.14159265358979323846 << endl;
```

Of course, in this case, a better solution is to just use **const**, which wasn't available in C. In C++, we discourage the use excessive use of the preprocessor because it can lead to unreadable, hard-to-maintain code, as well as hiding bugs beneath several layers of obfuscation. Of course, some of you may revel in that, so you'll want to look at the winners of the annual Obfuscated C Code Contest.

The one place where we still use **#define** in C++ is for the use of **source control** using conditional compilation. We'll look at that next.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Conditional Compilation

For **source code control**, you can **#define** a symbol and not give it a value. The preprocessor supports **conditional compilation**, using **preprocessor directives** to conditionally include a section of code based on which **#defined** symbols it has "seen":

```
#if defined(A)
    cout << "A is defined." << endl;
#elif defined(B)
    cout << "B is defined." << endl;
#else
    cout << "Neither A or B is defined." << endl;
#endif
```

This code is processed **before the rest of the code is sent to the compiler**; these conditions can **only** refer to **#defined** constants, integer values, and arithmetic and logical expressions using those values.

Here we've used the predefined preprocessor function **defined()** to check if a constant has been previously been **#defined**. You can also use these "shorthand" expressions.

- **#ifdef** is short for **if defined**; **#ifdef symbol** is the same as **#if defined(symbol)**.
- **#ifndef** is short for **if not defined**, the opposite of **#ifdef**.

Conditional compilation determines whether pieces of code are sent to the compiler. When the preprocessor encounters this, whichever conditional expression is true will have its corresponding code block included in the final program. If your code previously encountered **#define A**, then this entire portion of code will go to the compiler as:

```
cout << "A is defined." << endl;
```

The rest of the code will simply be discarded.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Predefined Symbols

There are several **predefined symbols** which your toolchain supplies, and which you can use in conditional compilation, like this example from StackOverflow, which tests for compiling on different platforms:

```
#ifdef _WIN32
    //define something for Windows (32-bit)
#elif __APPLE__
    // define something for OSX
#elif __linux
    // linux
#endif
```

These predefined symbols include those that are standard on every version of C++ and those that are common to GCC on every platform. There are also platform-specific symbols for other toolchains (such as the operating system). You can get a list of those by running **cpp -dM** from the shell.

For this problem, we care about is a **particular version of** C++. In the list of predefined standard constants, you'll see that **__cplusplus** (double leading underscores) contains version numbers for each release of C++. You can use that to bracket your own versions of the **stoi()** and **stod()** functions.

Go back to your test program and use this facility to define the functions **only** if the symbol **__cplusplus** is **<= 199711L**. Now you can compile and run with C++98 and with C++11/14/17/20 using the same source.

To implement the functions, just use code like this:

```
function stoi <- input str -> output int
    set result to 0
    construct an input string stream using str
    read from str into result
    return result
```

The **stod()** function will be identical, except **result** will be **double** instead of **int**.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Comparing the Results

Now, go ahead and you run the test program. Use **make 17** to compile and run under C++ 17, and **make 98** to compile and run under C++ 98. Your code should compile under both platforms. When you run it, however, **it doesn't produce exactly the same output** as it does under C++17, which uses the **stoi()** from the standard library.

```
cout << "stoi(\"42\")->" << stoi("42") << endl;
cout << "stod(\"3.14159\")->" << stod("3.14159") << endl;
cout << "stoi(\"3.14159\")->" << stoi("3.14159") << endl;
cout << "stod(\"4NonBlondes\")->" << stod("4NonBlondes") << endl;
cout << "stoi(\"UB-40\")->" << stoi("UB-40") << endl;
```

Look at the lines highlighted in yellow, where we pass **stod()** or **stoi()** **invalid input**. C++17 and C++98 produce the same output for the first four inputs, but the last one fails entirely. Neither the library nor your version fails on **stoi("3.14159")**. Both convert what they can (the **3**) and leaves the rest. But, the library version **crashes** with **stoi("UB-40")**; there is **no possible conversion**.



*So, that means the version we wrote is better, right?
After all, who wants a function that crashes?*

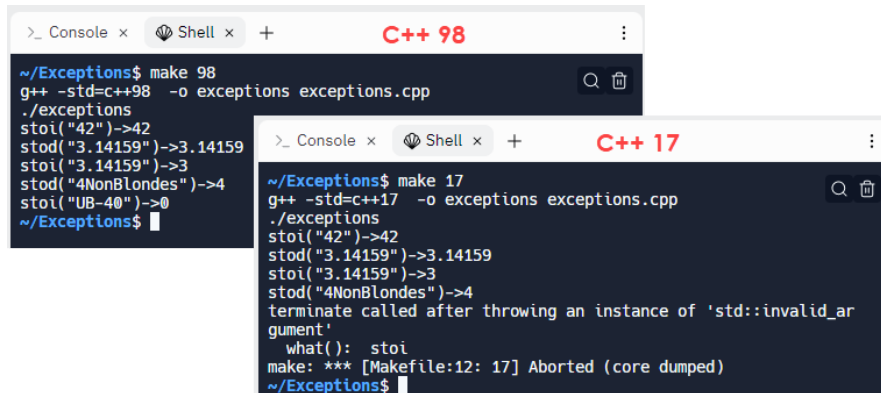
Well, not so fast. The question is, **what should stod()** and **stoi()** do with invalid input? In the next lesson, we'll use these techniques to look at more **error handling**.



This course content is offered under a CC Attribution Non-Commercial license.
Content in this course can be considered under this license unless otherwise noted.

Handling Errors

We finished the last lesson with a question: what **should** the `stoi()` and `stod()` functions do when given invalid input? The C++ 17 version, from the standard library, does one thing, while the version we wrote does something else entirely when given the invalid input "UB-40".



The image shows two terminal windows side-by-side. The left window is titled 'C++ 98' and shows the output of a program compiled with g++ -std=c++98. It demonstrates that for the input 'UB-40', both `stoi` and `stod` return 0. The right window is titled 'C++ 17' and shows the output of a program compiled with g++ -std=c++17. For the same input 'UB-40', the program terminates with an exception: `terminate called after throwing an instance of 'std::invalid_argument'`. The `what()` function shows the error was thrown from `stoi`. The `make` command output indicates the program was aborted (core dumped).

```
~/Exceptions$ make 98
g++ -std=c++98 -o exceptions exceptions.cpp
./exceptions
stoi("42")->42
stod("3.14159")->3.14159
stoi("3.14159")->3
stod("4NonBlondes")->4
stoi("UB-40")->0
~/Exceptions$
```

```
~/Exceptions$ make 17
g++ -std=c++17 -o exceptions exceptions.cpp
./exceptions
stoi("42")->42
stod("3.14159")->3.14159
stoi("3.14159")->3
stod("4NonBlondes")->4
terminate called after throwing an instance of 'std::invalid_ar
gument'
what():  stoi
make: *** [Makefile:12: 17] Aborted (core dumped)
~/Exceptions$
```

To answer this question, first consider what **should** happen when you try to:

- Print the square root of -2?
- Open a file that doesn't exist? Read data from that file?
- Convert a string that doesn't contain a number to a number?

Each of these is **handled in a different** way. None of them are syntax or linker errors. Instead, they are **runtime errors**. The compiler and linker produced an executable, but when it runs, an error occurs.

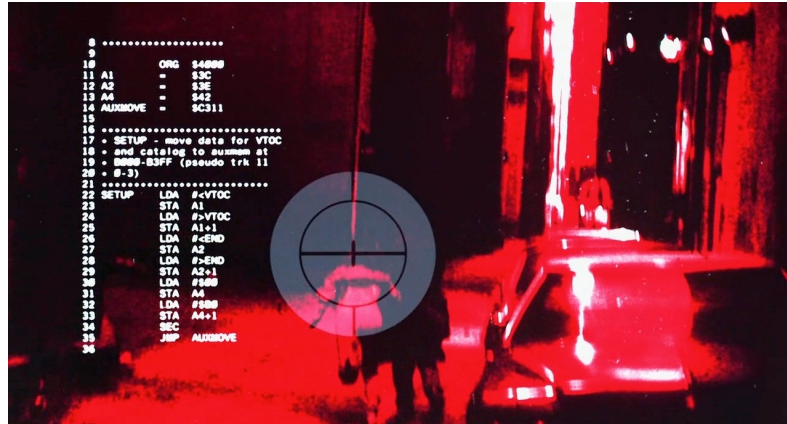
Let's examine a few ways to **handle** such errors.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

The Terminator

One option is to write a function like `die()`, which you saw in earlier lessons, and which prints an error message and then terminates. This is not really a good solution for runtime errors unless they are very severe, since it **doesn't give the user a chance to recover**. Imagine if your Web browser **shut down** every time you typed a URL incorrectly or clicked on a dead link.



Using a function like `die()` does prevent the program from continuing with garbage values, (which is good!), but it is simply too drastic and **too inflexible** to be a good universal approach.

However, there is one time when a "terminator" **is the correct way** to handle errors:

- When you are developing your code and ...
- When the error is a programming problem that you can fix.

In fact, the C++ library has a **built-in macro** which does this, called `assert()`.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Using assert

An **assertion** is a statement about a condition which must be true when encountered. If the condition is **not true**, then `assert()`, (declared in `<cassert>`), causes the program to immediately fail, printing an error message.

Programmers use assertions to **reason** about **logical correctness**. Assertions can be used to check **preconditions** (what must be true before the program runs correctly), and **postconditions** (what must be true after a calculation completes).

Here is an (admittedly silly) example using `assert()`.

```
cout << "Making sure that 2 + 2 is 5?" << endl;
assert(2 + 2 == 5); // false
```

The programmer assumed (**wrongly**) that the expression `2 + 2` should produce `5`. The assertion causes the program to stop and **print an error message**, so the programmer can fix the mistake. The message will depend on the toolchain. Here is `g++` on Unix.

```
Making sure that 2 + 2 is 5?
a.out: main.cpp:10: int main():
    Assertion `2 + 2 == 5' failed.
Aborted (core dumped)
```

The message includes the executable name (`a.out`), the source file (`main.cpp`), the line number (`10`), the function name and the assertion which failed, so you can immediately open your editor and fix the code.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

More Assertions

Assertions are **not** designed to handle runtime errors. They are designed to point out bugs in your code. **Steve Maguire**, one of the original developers of Excel, wrote a classic book named *Writing Solid Code*, which contains a chapter on assertions in C. Here are the points he makes:



- Assertions are shorthand way to write **debugging checks**
- Use assertions to check for **illegal conditions**, not error conditions
- Use assertions to **validate function arguments** under your control
- Use assertions to validate any **assumptions** you have made

If you want your code to **help you** find your bugs, make liberal use of `assert()`.

Since assertions are only needed while you are developing your code, you can remove them from your **production build** by compiling with the `-D NDEBUG` compiler switch, or by adding `#define NDEBUG` before including `<cassert>`.

`assert()` is not actually a function, but a **preprocessor macro**, so defining `NDEBUG` allows the preprocessor to remove all `assert()` statements before your code is compiled. Because of this, you need to make sure that an `assert()` never has a side effect, which could change the way your program runs when it is removed.

Static Asserts

C++ 11 also introduced the `static_assert()` declaration which may be used to double-check your assumptions about the platform you are developing on. For instance, if your code assumes that the `int` type is a 32-bit signed number, you can check that with:

```
static_assert(sizeof(int) == 4, "int must be 32 bits.");
```

Unlike regular assertions, `static_assert` is checked when you compile; it does not check for runtime errors. You can only check on compile-time constants and the error message must be a string literal; you cannot include variables. (In C++17 you may omit the error message.)



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Completion Codes

A second error-reporting option is the "tried-and-true" traditional **completion code technique** used for years in C, Pascal and FORTRAN. Have your function **return a special value** meaning that "the function failed to execute correctly."



In a way, this is what `sqrt()` does; it returns the "special" not-a-number value when its answer cannot be converted to a valid **double**. You can test for this value using the `isnan()` function in the header `<cmath>`. You could use the "error code" like this:

```
if (isnan(answer = sqrt(-1))) { /* error */ }
```

The `isnan()` function was added to C++ 11. Before that, `sqrt()` set the global variable `errno`, defined in `<cerrno>`, which was used like this.

```
double answer = sqrt(-1.0); // invalid
if (errno == EDOM) { /* invalid DOMain */ }
```



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Error Flags

With the advent of object-oriented programming, a variation on completion codes was birthed—**error state** which is encapsulated in objects. Of course, you've already encountered this with the input stream classes.

Here's an example. What happens if the user enters *twelve*?

```
cout << "Enter an integer: ";  
int n;  
cin >> n;    // Error state is set here
```

Each stream object has an **internal data member** that contains an individual error code, or **error flag**. These flags are given names like **badbit**, **goodbit** and **failbit**. If the user enters *twelve*, then the **failbit** is "set". If the keyboard isn't working, the **badbit** is set.

In the C-style of programming, you use **bitwise logical operators** (something we won't cover in this class, but you'll probably encounter in Computer Architecture) to read or set each of these error codes. In C++, however, you have **member functions**:

```
cin >> n;    // Error state may be set here  
if (cin.fail()) // Check if failbit is set  
{  
    cin.clear(); // clear all of the error flags  
    // empty the input stream and try again  
}
```

The big problem with completion codes and with error states, is that **you can ignore the return value without encountering any warnings**. Research has shown that programmers almost **never** check them. To better handle these kinds of problems, C++ introduced **exception handling**. If an error occurs inside a function, rather than returning a value, you report the problem and **jump to the proper error-handling code**.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.