

Dynamic Memory Allocation, Destructors

Dynamic Memory Allocation: `new` and `delete`

Overview

C++ allows **dynamic management of memory at run time** via two operators:

1. `new` – allocates memory for objects of any built-in or user-defined type
 - The amount of allocated memory depends on the object size
 - For user-defined types, size is determined by the data members
 - Memory is allocated in the free storage also known as **heap**, a random access memory (RAM) region assigned to each program at run time
2. `delete` – de-allocates memory used by `new` and frees it so the system can re-use it

```
pointer_variable = new data_type;  
delete pointer_variable;
```

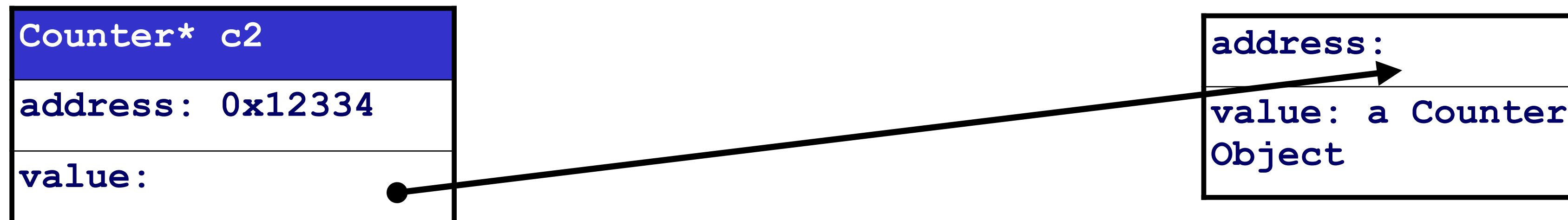
Dynamic Memory Allocation: new and delete

Example

Automatic variable
in the stack

```
Counter* c2 = new Counter("c2");  
delete c2; // de-allocate memory!
```

Dynamic object
in the heap



- `new` allocates an amount of memory given by `sizeof(Counter)` somewhere in memory
- It returns a pointer to this location
- We assign `c2` to be this pointer and access the dynamically allocated memory
- `delete` de-allocates that specific region of memory and makes it available again

Memory Leak: Killing the System

- One of the most common problems in (C++) programming
- Code allocates memory at run time with `new` but never releases it with a call to `delete`
- **Golden rule:** every time you call `new` ask yourself
 - Is using `new` Really necessary?
 - Where and when is `delete` called to free this memory?
- Even small amount of leak can lead to a system crash
 - E.g., leaking 10 kB per iteration in a 1M iterations loop leads to 1 GB of allocated an un-usable memory

Simple Example of Memory Leak

Based on `examples/05/Leak.cpp`

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

int main() {
    int i=1;
    while(i>0){

        double* ptr = new double[1000000000];
        ptr[0] = 1.1;

        cout << "i: " << i
              << ", ptr: " << ptr
              << ", ptr[0]: " << ptr[0]
              << endl;

        //delete[] ptr; // ops! memory leak!
        i++;
    }

    return 0;
}
```

- At each iteration, `ptr` is a pointer to a new (and large) array of 1G doubles
- This memory is not released because we forgot the delete operator
- Every round, more memory becomes unavailable, until the system runs out of memory and crashes

```
$ g++ -o Leak Leak.cpp
$ ./Leak
i: 1, ptr: 0x280000000, ptr[0]: 1.1
<...>
i: 17539, ptr: 0x7ffe150f8000, ptr[0]: 1.1
Leak(35944,0x100c3fd40) malloc: can't allocate region
*** mach_vm_map(size=8000012288, flags: 100) failed (error code=3)
Leak(35944,0x100c3fd40) malloc: *** set a breakpoint in malloc_error_break to debug
libc++abi.dylib: terminating with uncaught exception of type std::bad_alloc:
std::bad_alloc
Abort trap: 6
```

Pros and Cons of Dynamic Memory Allocation

- No need to fix at compilation time the size of data to be used
 - Easier to deal with real life use cases with variable and unknown number of data objects (e.g., the number of particles produced by a particle passing through a detector is unknown a priori)
 - No need to reserve very large, but FIXED-SIZE arrays of memory
- Main disadvantage: correct memory management
 - Must keep track of **ownership** of **objects**
 - Forgetting to de-allocate causes memory leaks, leading to slow execution and crashes
- Other disadvantage: can affect performance

Destructor Method of a Class

- Constructor used by compiler to initialise instance of a class (an object)
 - Allocate the object in memory and assign proper values to data members
- **Destructor**: special member function doing reverse work of constructors
 - Perform termination house keeping when objects go out of scope
 - No de-allocation of memory
 - Tell the program that memory previously occupied by the object is again free for use
 - FUNDAMENTAL when using dynamic memory allocation

Special Features of Destructors

Based on `examples/05/CounterDestructor.h`

```
// CounterDestructor.h
#ifndef Counter_h_
#define Counter_h_

#include <string>
#include <iostream>

using namespace std;

class Counter {
public:
    Counter(const std::string& name);
    ~Counter();
    int value();
    void reset();
    void increment(int step = 1);
    void print();

private:
    int count_;
    std::string name_;
};
#endif
```

- They have no arguments
- Like constructors, they do not have a return type
- Destructor of class Counter MUST be called
`~Counter()`

Example of Destructor

Based on `examples/05/CounterDestructor.*`

```
// CounterDestructor.h
#ifndef Counter_h_
#define Counter_h_

#include <string>
#include <iostream>

using namespace std;

class Counter {
public:
    Counter(const std::string& name);
    ~Counter();
    int value();
    void reset();
    void increment(int step = 1);
    void print();

private:
    int count_;
    std::string name_;
};
#endif
```

Constructor initializes data members

Destructor does nothing

```
// CounterDestructor.cc
#include "CounterDestructor.h"

#include <iostream>
using std::cout;
using std::endl;

Counter::Counter(const std::string& name) {
    count_ = 0;
    name_ = name;
    cout << "Counter::Counter() called for Counter "
         << name_ << endl;
};

Counter::~~Counter() {
    cout << "Counter::~~Counter() called for Counter "
         << name_ << endl;
};

int Counter::value() { return count_; }

void Counter::reset() { count_ = 0; }

void Counter::increment(int step) {
    count_ = count_ + step;
}

void Counter::print() {
    cout << "Counter::print(): name: " << name_
         << " value: " << count_ << endl;
}
```

Who and Calls the Destructor and When?

Based on examples/05/Destructor1.cpp

```
#include "CounterDestructor.h"
#include <string>

int main() {

    Counter c1( std::string("c1") );
    Counter c2( std::string("c2") );
    Counter c3( std::string("c3") );

    c2.increment(135);
    c1.increment(5677);

    c1.print();
    c2.print();
    c3.print();

    return 0;
}
```

- Compiler calls constructors when new objects are created
- Compiler implicitly calls destructors when objects go out of scope
- Destructors are called in reverse order of creation (c1, c2, c3 vs. c3, c2, c1)

```
$ g++ -o Destructor1 Destructor1.cpp CounterDestructor.cc
$ ./Destructor1
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::print(): name: c1  value: 5677
Counter::print(): name: c2  value: 135
Counter::print(): name: c3  value: 0
Counter::~~Counter() called for Counter c3
Counter::~~Counter() called for Counter c2
Counter::~~Counter() called for Counter c1
```

Who and Calls the Destructor and When?

Based on `examples/05/Destructor2.cpp`

```
#include "CounterDestructor.h"
#include <string>

int main() {
    Counter c1( std::string("c1") );

    int count = 344;

    if( 1.1 <= 2.02 ) {
        Counter c2( std::string("c2") );

        Counter c3( std::string("c3") );

        if( count == 344 ) {
            Counter c4( std::string("c4") );
        }

        Counter c5( std::string("c5") );

        for(int i=0; i<3; ++i) {
            Counter c6( std::string("c6") );
        }
    }

    return 0;
}
```

- Compiler calls constructors when new objects are created
- Compiler implicitly calls destructors when objects go out of scope
- Destructors are called in reverse order of creation



What will the output be?

Who and Calls the Destructor and When?

Based on examples/05/Destructor2.cpp

```
#include "CounterDestructor.h"
#include <string>

int main() {
    Counter c1( std::string("c1") );

    int count = 344;

    if( 1.1 <= 2.02 ) {
        Counter c2( std::string("c2") );

        Counter c3( std::string("c3") );

        if( count == 344 ) {
            Counter c4( std::string("c4") );
        }

        Counter c5( std::string("c5") );

        for(int i=0; i<3; ++i) {
            Counter c6( std::string("c6") );
        }
    }

    return 0;
}
```

```
$ g++ -o Destructor2 Destructor2.cpp CounterDestructor.cc
$ ./Destructor2
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::Counter() called for Counter c4
Counter::~~Counter() called for Counter c4
Counter::Counter() called for Counter c5
Counter::Counter() called for Counter c6
Counter::~~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~~Counter() called for Counter c6
Counter::~~Counter() called for Counter c5
Counter::~~Counter() called for Counter c3
Counter::~~Counter() called for Counter c2
Counter::~~Counter() called for Counter c1
```

new, delete, and Destructors

Based on examples/05/Destructor3.cpp

```
#include "CounterDestructor.h"
#include "../04/Datum.h"
#include <iostream>
using namespace std;

int main() {

    Counter c1("c1");

    Counter* c2 = new Counter("c2");
    c2->increment(6);


    Counter* c3 = new Counter("c3");

    Datum d1(-0.3,0.07);

    Datum* d2 = new Datum( d1 );
    d2->print();

    delete c2; // de-allocate memory!
    delete c3; // de-allocate memory!
    delete d2;

    return 0;
}
```

 What do these do?

```
$ ln -s ../04/Datum.h Datum.h
$ ln -s ../04/Datum.cc Datum.cc
$ g++ -o Destructor3 Destructor3.cpp CounterDestructor.cc Datum.cc
$ ./Destructor3
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
datum: -0.3 +/- 0.07
Counter::~~Counter() called for Counter c2
Counter::~~Counter() called for Counter c3
Counter::~~Counter() called for Counter c1
```

- Compiler calls constructors when new objects are created, with and without usage of new
- delete calls destructors explicitly to de-allocate memory
- Compiler calls destructor implicitly when objects go out of scope



Why no message for d2?

g++ Options, External Libraries, Command Line Arguments, File I/O

Options of g++

```
$ man g++
```

```
GCC(1)
```

```
GNU
```

```
GCC(1)
```

```
NAME
```

```
gcc - GNU project C and C++ compiler
```

```
SYNOPSIS
```

```
gcc [-c|-S|-E] [-std=standard]
    [-g] [-pg] [-Olevel]
    [-Wwarn...] [-Wpedantic]
    [-Idir...] [-Ldir...]
    [-Dmacro[=defn]...] [-Umacro]
    [-foption...] [-mmachine-option...]
    [-o outfile] [@file] infile...
```

Only the most useful options are listed here; see below for the remainder. g++ accepts mostly the same options as gcc.

```
DESCRIPTION
```

When you invoke GCC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the -c option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command-line options that you can use with GCC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

The gcc program accepts options and file names as operands. Many options have multi-letter names; therefore multiple single-letter options may not be grouped: -dv is very different from -d -v.

Some already familiar options

- **-o**: specify name of the output (default is a.out for binary executable)
- **-E**: stop after running pre-compiler to resolve pre-compiler directives (do not compile nor link the binary)
- **-c**: stop after compilation (do not link so no executable is produced)

Increasing Warning Level

Based on `examples/05/AllWarnings.cpp`

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

int index() {
    int i = 27;
}

std::string name() {
    std::string str("test of g++ options");
    return str;

    // text after return
    int j = 56;
}

int main() {
    int i = index();
    std::string st = name();
    std::cout << "i:" << i
              << "\t st: " << st
              << std::endl;

    return 0;
}
```

Very often simple warnings are a clear sign of something seriously wrong...

```
$ g++ -o AllWarnings AllWarnings.cpp
AllWarnings.cpp:7:1: warning: non-void function does not return a value [-Wreturn-type]
}
^
1 warning generated.
```

```
$ g++ -o AllWarnings AllWarnings.cpp -Wall
AllWarnings.cpp:6:7: warning: unused variable 'i' [-Wunused-variable]
    int i = 27;
    ^
AllWarnings.cpp:7:1: warning: non-void function does not return a value [-Wreturn-type]
}
^
AllWarnings.cpp:14:7: warning: unused variable 'j' [-Wunused-variable]
    int j = 56;
    ^
3 warnings generated.
$ ./AllWarnings
i:1          st: test of g++ options
```

...value of `index()` is 1! Ignores completely the implementation!

Optimizing Your Executable with g++

g++ offers many options to optimize your executable and reduce execution time

- Compiler analyzes your code to determine the best execution path
- Compiling with optimization takes longer
- Optimized program is harder to debug
- Results of optimized and non-optimized executables MUST be identical

We will not go into the details, but optimization is extremely important: the resources and time that separate you from results are precious

Options That Control Optimization

These options control various sorts of optimizations.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you expect from the source code.

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them.

Not all optimizations are controlled directly by a flag. Only optimizations that have a flag are listed in this section.

Most optimizations are only enabled if an -O level is set on the command line. Otherwise they are disabled, even if individual optimization flags are specified.

Depending on the target and how GCC was configured, a slightly different set of optimizations may be enabled at each -O level than those listed here. You can invoke GCC with -Q --help=optimizers to find out the exact set of optimizations that are enabled at each level.

Levels of Optimization

`-O`

`-O1` Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With `-O`, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

`-O` turns on the following optimization flags:

`-fauto-inc-dec -fcompare-elim -fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fguess-branch-probability -fif-conversion2 -fif-conversion -fipa-pure-const -fipa-profile -fipa-reference -fmerge-constants -fsplit-wide-types -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp -ftree-ch -ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-phirop -ftree-slsr -ftree-sra -ftree-pta -ftree-ter -funit-at-a-time`

`-O` also turns on `-fomit-frame-pointer` on machines where doing so does not interfere with debugging.

`-O2` Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to `-O`, this option increases both compilation time and the performance of the generated code.

`-O2` turns on all optimization flags specified by `-O`. It also turns on the following optimization flags: `-fthread-jumps -falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize -fexpensive-optimizations -fgcse -fgcse-lm -fhoist-adjacent-loads -finline-small-functions -findirect-inlining -fipa-sra -foptimize-sibling-calls -fpartial-inlining -fpeephole2 -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec -fschedule-insns -fschedule-insns2 -fstrict-aliasing -fstrict-overflow -ftree-switch-conversion -ftree-tail-merge -ftree-pre -ftree-vrp`

Please note the warning under `-fgcse` about invoking `-O2` on programs that use computed gotos.

`-O3` Optimize yet more. `-O3` turns on all optimizations specified by `-O2` and also turns on the `-finline-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-after-reload`, `-ftree-vectorize`, `-fvect-cost-model`, `-ftree-partial-pre` and `-fipa-cp-clone` options.

`-O0` Reduce compilation time and make debugging produce the expected results. This is the default.

`-Os` Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

`-Ofast`

Disregard strict standards compliance. `-Ofast` enables all `-O3` optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on `-ffast-math` and the Fortran-specific `-fno-protect-parens` and `-fstack-arrays`.

`-Og` Optimize debugging experience. `-Og` enables optimizations that do not interfere with debugging. It should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

If you use multiple `-O` options, with or without level numbers, the last such option is the one that is effective.

CAUTION! Extremely dangerous
when precision is a goal.

Debug Symbols with -g

- Produce debugging information to be used by debuggers, for example GDB: yields a larger binary
- Extremely useful when first developing code: shows high level labels (function names and variables)
- It slows down the code a bit, but may pay off in development phase
- Once code fully tested, remove this option and fully optimize

`-g` Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2). GDB can work with this debugging information.

On most systems that use stabs format, `-g` enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but probably makes other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use `-gstabs+`, `-gstabs`, `-gxcoff+`, `-gxcoff`, or `-gvms` (see below).

GCC allows you to use `-g` with `-O`. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values are already at hand; some statements may execute in different places because they have been moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

The following options are useful when GCC is generated with the capability for more than one debugging format.

Libraries of Compiled Code

Libraries are simple archives that contain compiled code (object files)

1. **Static Library**: used by linker to include compiled code in the executable at linking time
 - Larger executable since it includes ALL binary code run during execution
 - Does not require presence of libraries at runtime since all code already included in the executable
2. **Shared Library**: used by executable at runtime (we may discuss them in a future lecture)
 - The binary holds only references to functions in the libraries but the code is not included in the executable itself
 - Smaller executable size but **REQUIRES** library to be available at runtime

Creating and Using Static Libraries

Based on `examples/05/StaticLib/`

```
$ g++ -c Datum.cc
$ g++ -c InputService.cc
$ g++ -c Calculator.cc
```

Compile library components

```
$ ar -r libMyLib.a Datum.o InputService.o Calculator.o
ar: creating libMyLib.a
```

Create archive of compiled codes
Note the `lib` prefix

```
$ ar -tv libMyLib.a
rw-r--r--      502/20      1664 Sep 14 21:35 2021 __.SYMDEF SORTED
rw-r--r--      502/20     15352 Sep 14 21:35 2021 Datum.o
rw-r--r--      502/20      3496 Sep 14 21:35 2021 InputService.o
rw-r--r--      502/20      1944 Sep 14 21:35 2021 Calculator.o
```

```
$ g++ -o wgtavg wgtavg.cpp -l MyLib -L .
$ ./wgtavg
weighted average: 1 +/- 0.1
```

Compile and link main code
`-l <library name without lib prefix and .a extension>`
`-L < path to library>`

Common g++ Options for External Libraries

- When using external libraries you are usually provided with
 - path to directory where you can find include files
 - path to directory where you can find libraries
 - NO access to source code! But source code is not needed to compile, only header files: only interface matters!
- **-L**: path to directory containing libraries (e.g., `-L /usr/local/root/5.08.00/lib`)
- **-I**: path to directory containing header files (e.g., `-I /usr/local/root/5.08.00/include`)
- **-l**: specify name of libraries to be used at link time, omitting prefix `lib` and extension `.a` (e.g., `-l Core -lHbook`, with and without the space)

Passing Arguments to C++ Applications

Based on `examples/05/Args.cpp`

- `argc` is number of command line arguments: it includes the application name as well
- `argv` is vector of pointers to characters: interprets each set of disjoint characters as a token

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

int main(int argc, char* argv[]) {

    cout << "# of cmd line arguments argc: " << argc << endl;
    cout << "argv[0]: " << argv[0] << endl;
    cout << "Running " ;
    for(int i=0; i<argc; i++){
        cout << argv[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```
$ g++ -o Args Args.cpp
$ ./Args
# of cmd line arguments argc: 1
argv[0]: ./Args
Running ./Args

$ ./Args jello world
# of cmd line arguments argc: 3
argv[0]: ./Args
Running ./Args jello world
```

Handling Non-string Values

- `argv` is a pointer to characters, but we may want to receive (and process) number arguments
 - `atoi` converts char to int
 - `atof` converts char to double
- The developer carries the **responsibility of checking the validity of arguments** provided at runtime

```
if(argc < 4 ) {  
    //Fail and print an error message that is USEFUL for the user!  
    return -1; // can be used by user to determine error condition  
}
```

**“Not enough arguments” is NOT good enough! How many should they be?
What is each argument for? And remember to provide examples for users**

ifstream and sscanf

Based on examples/05/ReadFile.cpp

Input from file with ifstream

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {

    // file name
    const char filename[30] = "input.txt";

    // create object for input file
    ifstream infile(filename); //input file object

    // string to hold each line
    string line;

    // make sure input file is open otherwise exit
    if(!infile.is_open()) {
        cerr << "cannot open input file" << endl;
        return -1;
    }

    // CONTINUES
```

Parsing input lines with sscanf

```
// CONTINUED

// variables to read in from file at each line
char name[30];
double val, errpos;
float errneg;

// loop over file until end-of-file
while(!infile.eof()) {
    // get current line
    getline(infile, line);
    if( line == "\n" || line == "" ) continue;

    // parse line with the provided format and put data in variables
    // NB: USING POINTERS TO VARIABLES
    // format: %s string    %f float    %lf double
    sscanf(line.c_str(), "%s %lf %lf %f", name, &val, &errpos, &errneg);

    // print out for debug purposes
    cout << "name: " << name
         << "\t\tvalue: " << val << "\t\tpos err: " << errpos
         << "\t\tneg err: " << errneg << endl;
} // !eof

infile.close(); // close input file before exiting

return 0;
}
```

ofstream to Write to File

Based on examples/05/ReadWriteFile.cpp

```
// variables to read in from file at each line
char name[30];
double val, errpos;
float errneg;
```

```
// create object for output file
ofstream outfile;
```

```
// output file
outfile.open ("output.txt");
outfile << "Writing this to a file.\n";
```

```
// loop over file until end-of-file
while(!infile.eof()) {
    // get current line
    getline(infile,line);
    if( line == "\n" || line == "" ) continue;

    // parse line with the provided format and put data in variables
    // NB: USING POINTERS TO VARIABLES
    // format: %s string    %f float    %lf double
    sscanf(line.c_str(),"%s %lf %lf %f", name, &val, &errpos, &errneg);

    // print out for debug purposes
    cout << "name: " << name
         << "\t\tvalue: " << val << "\t\tpos err: " << errpos
         << "\t\tneg err: " << errneg << endl;

    // print to file
    outfile << "name: " << name
             << "\t\tvalue: " << val << "\t\tpos err: " << errpos
             << "\t\tneg err: " << errneg << endl;
} // !eof

infile.close(); // close input file before exiting
outfile.close(); // close output file before exiting

return 0;
}
```