CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 1
August 29, 2018

About This Course

Where to find information

Information about this course is posted on the course website:

https://zoo.cs.yale.edu/classes/cs427/2018f/

- Syllabus.
- One of the online textbooks, Exploring C++ by Alice Fischer.
- Lecture notes.
- Code samples.
- Homework assignments.

The course uses Canvas for assignments and announcements. It also contains some links to the main course website on the Zoo.

The syllabus contains important additional information. Read it!

Course mechanics

You will need a Zoo course account. It should be created automatically when you register for this course as a Shopper or Student. The login credentials will be your standard Yale NetID and password. Test it now!

Assignments will be submitted on Canvas using the Assignments tool. Detailed instructions will be provided.

Course Requirements: Homework assignments (∼40%), midterm exam (∼20%), final exam (∼40%).

Course goals

Learn how to answer the following questions:

1. *Who* programs and why?
2. *How long* does a program last?
3. *What* are the characteristics of a good program?
4. *When* do good programs matter?
5. *How* does C++ help one write good programs?

Discussion.

Who programs and why?

People program for different reasons.

1. To get answers to particular problems of interest.
2. To avoid repetitive work when solving several instances of the same problem.
3. To provide tools that others can use.
4. To produce software of commercial value.
5. To provide a mission-critical service.

How long does a program last?

Three facetious answers:

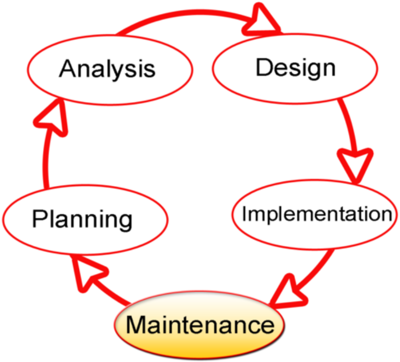
1. Until it stops being useful.
2. Until nobody maintains it.
3. Far longer than was originally anticipated.

What are the characteristics of a good program?

1. **Correctness:** Does what is intended.
2. **Robustness:** Handles bad input gracefully.
3. **Security:** Resists malicious exploits.
4. **Efficiency:** Makes cost-effective use of computer resources.
5. **Isolation:** Prevents unintended interactions within itself and with its hardware and software environment.
6. **Cleanliness:** Embodies a direct connection between the task and the solution.
7. **Clarity:** Can be comprehended rapidly by humans.
8. **Maintainability:** Has complete test suite. Modifications cause expected changes to behavior.

When do good programs matter?

The program development life cycle is a continuous circular process:



By Dzonatas (Own work) [CC BY-SA 3.0 or GFDL], via Wikimedia Commons

Important imperatives for life cycle management

1. Modularity – group related parts together at each level
2. Non-interference – protect unrelated parts from each other
3. Produce clean, simple, straightforward, understandable code
4. Avoid replicated code fragments
5. Avoid unnecessary hardware and OS dependencies
6. Follow recognized style guidelines
7. Document your work approrpiately

How does C++ help one write good programs?

1. Language and core library are standardized and documented.
2. Classes, functions and templates support modularity.
3. Privacy and const attributes protect and isolate code.
4. Constructors/destructors ensure object coherence.
5. Inheritance and templates help one avoid replicated code.
6. Exceptions separate error handling from normal program flow.
7. Operator extensions and qualified names improve readability.
8. Inline functions, const, reference types, move semantics, stack-allocated objects, and static type checking permit better code efficiency.

Topics to be Covered

Major Areas

1. Foundations of C++ (basics of objects and classes).
2. Software toolset.
3. C++ storage model: paradigms for object creation and deletion, pointers, references, lvalues and rvalues, move semantics.
4. Software design process
5. Programming for reliability, testing, debugging.
6. Programming for efficiency.

Course goals - practical

- Learn how to follow instructions, and how to question them if you think they are wrong.
- Learn how to get a big job done one module at a time.
- Learn how to use a reference manual.
- Learn how to design for efficiency and reliability.
- Learn how to test, analyze, and debug code.
- Learn how to present your work in a professional manner.
- Become proficient at C++ programming, starting with a knowledge of C.

Course goals - conceptual

- Learn what object-oriented programming is – and isn't.
- Learn what constitutes good object oriented design.
- Learn how C++ differs in syntax and semantics from standard ISO C
- Learn how C++ provides better support OO-programming than other object-oriented languages such as Python, Ruby, and Java.
- Learn about classes, objects, type hierarchies, virtual functions, templates, and their implementations in C++.
- Learn the principles behind the exception handler and when and how to use it.
- Learn how to use the standard C++ class libraries.

Kinds of Programming

Problem solving

Desired properties of programs for solving problems:

- Correct outputs from correct inputs
- Succinct expression of algorithm
- Simple development cycle

Beginning programming courses tend to focus on programs to solve small problems.

Industrial-Strength Sofware

- Thousands of lines of code
- Written by many programmers
- Over a large span of time
- Deployed on a large number of computers
- Used by many people
- With different architectures and operating systems
- Interacts with foreign code and devices
- Evolves over time

Software Construction

Desired properties of **industrial-strength** software:

- Correct outputs from correct inputs
- Robust in face of bad inputs; stable; resilient
- Economical in resource usage (time and space)
- Understandable and verifiable code
- Secure
- Easily repurposed
- Easily deployed
- Maintainable

Why C++?

C/C++ are popular

According to the TIOBE Index[1] for August 2018, C and C++ are the 2nd and 3rd most popular programming languages, behind only Java.

[1] See TIOBE Index

C/C++ is flexible

A typical software system is built in layers on top of the raw hardware:

5 Application
4 Application support (libraries, databases)
3 Virtual machine [optional]
2 Operating system
1 System kernel
0 Hardware

C/C++ are almost universally used to implement code at levels 1-4. Java is popular for levels 5, but recent additions to C++ make it increasingly attractive for level 5 applications as well.

Advantages and disadvantages of C++

- C++ allows one to construct stable, reliable, industrial-strength software.
- Many programming errors are detected by the compiler, resulting in reduced debugging time after the first successful compile.
- C++ is "closer" to the machine, making it possible to have better control over resource usage.

Downsides of C++

- C++ is a big powerful tool that can easily be misused.
- The C++ programmer must pay attention to how memory is managed. Mistakes in memory management can lead to catastrophic failures and security holes.
- C++ programs may be longer than other languages because the programmer learns to describe her program more fully.

C++ Programming Standards

Five commandments for this course

From Chapter 1 of Exploring C++ and elsewhere:

1. Use C++ input and output, not C I/O, for all assigned work.
2. Don't use global variables – you will lose points. If you think you need one, ask for help. Your class design is probably defective.
3. Don't use getter and setter functions. Rather, provide a public interface with semantically meaningful functions for querying and updating the state of an object.
4. Don't believe a lot of the rules of thumb you may have learned in a Java course or that you read on the internet. Java is different from C++ in many important ways, and many Java books do not focus on industrial strength programming.

Can is not the same as should!

From Chapter 1 of Exploring C++:

- C++ is a very powerful language, which, if used badly can produce projects that are badly designed, badly constructed, and impossible to debug or maintain.
- Your goal is to learn to use the language well, and with good style.
- Please read *and follow* the style guidelines in Section 1.2.
- Download the two tools files from the website.
- Read Section 1.3, about the tools library, and use this information to customize your own copy of the tools.

Rules for preparing your work

1. Every code file you submit must contain a comment at the top giving the name of the file, your name and netID, the course number, and the assignment number.
2. If your work is based on someone else's work, you *must* cite them at the top of the file and describe what part(s) of the code are theirs.
3. If you have started from a file that you obtained from someone else and it contains authorship/copyright information, you must leave that information in place.
4. If you have any doubts about the proper way to cite your sources, *ask*, don't just guess. Stay out of trouble.

Rules for submitting your work

1. All submissions must be done on Canvas.
2. Test every line of code you write. It is your job to verify that your entire program works. If you submit a program without a test plan and test output, the grader will assume that it does not compile and will grade it accordingly.
3. Compile and test your program on the Zoo before submission.
4. Supply a `Makefile` with your code so that a grader can type `make` and your code will compile and be ready to run.
5. Supply a `README` file that contains instructions to the grader on how to run and test your code.
6. Submit all files needed to compile your program, including copies files that have been provided for your use.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 2
August 31, 2018

Tasks for this week

- Log in to the Zoo. You may see a CPSC 427 subdirectory already created for you.
- Read Chapters 1–3 of *Exploring C++*. (36 pages in all.)
- Do problem set 1.

C++ Overview

Why did C need a ++?

Chapter 2 of Exploring C++

1. C was designed and constructed a long time ago (1971) as a language for writing Unix.
2. The importance of data modeling was poorly understood at that time.
3. Data types were real, integer, character, and array, of various sizes and precisions.
4. It was important for C to be powerful and flexible but not to have clean semantics.
5. Nobody talked much about portability and code re-use at that time.

Today, we demand much more from a language.

C++ was Designed for Modeling

Design goals for C++ (Bjarne Stroustrup)

1. Provide classes (replacing structs) as a means to model data.
2. Let a class encapsulate data, so that its implementation is hidden from a client program.
3. Permit a C++ program to link to libraries from other languages, especially FORTRAN.
4. Produce executable code that is as fast as C, unless run-time binding is necessary.
5. Be fully compatible with C, so that C programs could be compiled under a C++ compiler and still work properly.

General properties of C++

- Widely used in the real world.
- Close to the machine and capable of producing efficient code.
- Gives a programmer fine control over the use of resources.
- Supports the object-oriented programming paradigm.
- Supports modularity and component isolation.
- Supports correctness through privacy, modularity, and use of exceptions.
- Supports reusabale code through derivation and templates.

C++ Extends C

- C++ grew out of C.
- Goals were to improve support for modularity, portability, and code reusability.
- Most C programs will compile and run under C++.
- C++ replaces several problematic C constructs with safer versions.
- Although most old C constructs will still work in C++, several should *not* be used in new code where better alternatives exist.

  Example: Use Boolean constants `true` and `false` instead of 1 and 0.

Some Extensions in C++

- One-line comments //.
- Executable declarations.
- Type `bool`.
- Enumeration constants are no longer synonyms for integers.
- Reference types.
- Definable type conversions and operator extensions.
- Functions with multiple methods.
- Classes with private parts; class derivation.
- Class templates.
- An exception handler.

Building a Project

Modules

A **compilation module** is a collection of **header files** (`.h` or `.hpp`) and an **implementation file** (`.c` or `.cpp`) that can be processed by the C or C++ **compiler** to produce an **object file** (`.o`) file.

A **project** is a collection of compilation modules that can be processed by the **linker** to produce a runnable piece of code called an **application** (or **program** or **executable** or **command**).

Some modules are part of the project. Others come from **libraries** (`.a` or `.so` files) that contain object code for modules written by others and provided by the system for your use.

Whatever the origin of the modules, they must be joined together during final assembly to produce the runnable application. This step of the process is called **linking**.

Separate compilation model

Unlike some languages, C/C++ permits independent compilation of modules. In the traditional **separate compilation model**, each module is compiled separately to produce a corresponding object file. Then the object files and necessary libraries are linked together to produce the executable.

The C/C++ programmer must clearly distinguish between compilation and linking, especially when interpreting error comments from the build process.

The build process

To summarize, the process of building an executable file consists of two phases:

1. Each module in the project is compiled to produce corresponding object files.
2. All object files in the project are linked together with necessary libraries to produce the executable file.

Because the executable must be rebuilt every time one of the source files is changed, manually going through the build process can be tedious and error-prone.

Automating the build process

Two common ways to automate the build process:

1. Use the `make` command. `make` reads a special file (`Makefile` or `makefile`) which contains a description of the necessary steps to build the application. It's also smart about not recompiling modules that have not changed since the last build.
2. Use an Integrated development environments (IDE) such as `Xcode` on the Mac or `Eclipse` on linux machines. The IDE keeps track of which modules belong to the project so that they can be rebuilt when needed.

Local build requirement

In this course, you're free to use whatever build tools you wish. However, you must submit a correct makefile as part of your code so that the grader can simply type `make` in order to produce an executable that will run on the Zoo.

What comprises a module?

A module consists of one or more **header files** and at most one **implementation** file.

Header files provide the context to the compiler for understanding the code in the implementation file. The `#include` directive names a header file that the compiler should process when compiling this module.

Header files for system libraries are often found in the `/usr/include` directory, but they can be put anywhere as long as the *compiler* is told where to look for them.

Header files for the current module are generally located in the same directory as the implementation file being compiled.

Header files

Header files contain class, data, function, and other declarations that are needed by the **client** of the module. They need to be included by every module that uses those declarations. Header files must not contain executable code. Doing so can lead to obscure multiply-defined errors at link time.

There is no uniform naming convention for header files. In C, people generally use the `.h` file name extension. For C++, some people continue to use `.h`. This often works okay, but it can lead to problems with projects that mix modules written in C with those written in C++.

An unambiguous convention is to restrict `.h` to C header files and to use `.hpp` for C++ header files. *We will use that convention in this course.*

What's in an implementation file?

Implementation (`.cpp`) files contain **definitions** of functions and constants that comprise the actual runnable code.

Each compiled definition must appear in exactly one object file. If it appears in more than one, the linker will generate a multiply-defined error.

For this reason, *definitions* are never put in header files.[1]

[1] Template classes are an exception to this rule, but for non-obvious reasons deriving from how the compiler handles templates.

Compiling in linux

The Zoo machines have two different C++ compilers installed: `g++` and `clang++`. Both are good compilers.

`g++` is the venerable Gnu C++ compiler. It is fast and generally very good.

`clang++` is a newer, more modular, compiler. It is slower to run than `g++` but sometimes may give better object code. It also gives different error messages which sometimes are clearer than those from `g++` (and sometime they are less clear).

You may find both compilers useful in developing your code. However, the final result must run using `g++`, and your `makefile` must be written to ensure that `g++` will be used.

Invoking the compiler

`g++` and `clang++` are commands used to invoke the corresponding compilers. However, depending on the command line switches given, they can be instructed to compiler and/or link several modules with one invocation.

For example,

```
g++ -o mycommand mod1.cpp mod2.cpp mod3.cpp
```

will compile all three `.cpp` files and then link the results togeter to produce an executable file `mycommand`. On the other hand, when used with the `-c` switch,

```
g++ -c -o mod1.o mod1.cpp
```

compiles the one module `mod1.cpp` to produce the single object file `mod1.o`.

Linking

When used without the `-c` switch, `g++` calls the linker `ld` to build an executable.

- If all command line arguments are object files, `g++` just does the linking.
- If one or more `.cpp` files appear on the command line, `g++` first compiles them and then links the resulting object files together with any `.o` files given on the command line. In this case, `g++` combines compilation and linking, and it does not write any new object files.

In both cases, the linker completes the linking task by searching libraries for any missing (unresolved) functions and variables and linking them into the final output.

System libraries

System libraries are often found in directories `/lib`, `/lib64`, `/usr/lib`, or `/usr/lib64`, but they can be placed anywhere as long as the *linker* is told where to find them.

The linker knows where to find the standard system libraries, and it searches the basic libraries automatically. Many other libraries are not searched unless specifically requested by the `-L` and `-l` linker flags.

One-line compilation

Often all that is required to compile your code is the single command

```
g++ -o myapp -O1 -g -Wall -std=c++17 *.cpp
```

The switches have the following meanings:

- `-o` name the output file;
- `-O1` do first-level optimization (which improves error detection);
- `-g` add symbols for use by the debugger;
- `-Wall` gives all reasonable warnings;
- `-std=c++17` tells the compiler to expect code in the C++17 language dialect.

The job of the project manager

As we've seen, a project consists of many different files. Keeping track of them and remembering which files and switches to put on the command line can be a major chore.

Project maintenance tools such as `make` and **Integrated Development Environments (IDEs)** are used to aid in this task.

Command line development tools

At the very least, you should become familiar with the basic tools for maintaining and building projects:

- A text editor such as `emacs` or `vi`.
- The compiler suite `g++`.
- The project manager `make`.

`clang++` is a newer alternative to `g++`. There are indications that it produces slightly better error messages and slightly better code than `g++`, but both compilers are very good and are suitable for use in this course. (The MacIntosh Xcode development system now defaults to `clang++`.)

Parts of a simple project

- Header file: `tools.hpp`
- Implementation files: `main.cpp`, `tools.cpp`
- Object files: `main.o`, `tools.o`
- Executable: `myapp`

Object files are built from implementation files and header files.

The executable is built from object files.

The `Makefile` describes how.

Dependencies

Whenever a source file is changed, the object files and executables that are directly or indirectly produced from it become out of date and must be rebuilt. Those files are called **dependencies** of the source file.

`make` uses dependency information stored in `Makefile` to avoid rebuilding files that have *not* changed since the last build. It only recompiles and/or relinks those files that are older than a file that they depend on.

`make` uses file modification dates for this purpose, so if those dates are off, `make` might fail to rebuild a file that is actually out of date.

A sample Makefile

```
#-------------------------------------------------------------
# Macro definitions
CXXFLAGS = -O1 -g -Wall -std=c++17
OBJ = main.o tools.o
TARGET = myapp
#-------------------------------------------------------------
# Rules
all: $(TARGET)
$(TARGET): $(OBJ)
	$(CXX) -o $@ $(OBJ)
clean:
	rm -f $(OBJ) $(TARGET)
#-------------------------------------------------------------
# Dependencies
main.o: main.cpp tools.hpp
tools.o: tools.cpp tools.hpp
```

Parts of a Makefile

A Makefile has three parts:

1. Macro definitions.
2. Rules.
3. Dependencies.

Syntax peculiarities:

- Lines beginning with `#` are comments.
- Indented lines must start with a `tab` character.

Macros

```
CXXFLAGS = -O1 -g -Wall -std=c++17
OBJ = main.o tools.o
TARGET = myapp
```

Macros are named strings.

- `CXXFLAGS` is added to the `g++` command line in **implicit rules**. Here we want level-1 optimization, symbols for the debugger, all warnings, and dialect c++17.
- `OBJ` lists the object files for our application.
- `TARGET` lists the final product (command).

Rules

```
all: $(TARGET)
$(TARGET): $(OBJ)
        $(CXX) -o $@ $(OBJ)
clean:
        rm -f $(OBJ) $(TARGET)
```

Rules tell how to build product files.

1. To build `all`, first build everything listed in `TARGET`.
2. To build `TARGET`, first build the `.o` files in `OBJ`. Then call the linker to create `TARGET` from the files in `OBJ`.
3. To build `clean`, generated files, delete everything in `OBJ` and `TARGET`.

Rules

```
all: $(TARGET)
$(TARGET): $(OBJ)
        $(CXX) -o $@ $(OBJ)
clean:
        rm -f $(OBJ) $(TARGET)
```

Notes:

- `CXX` is predefined to be the system default C++ compiler.
- `$@` is a special macro that refers the target of the current rule (`myapp` in the above example).
- `$(`*name*`)` refers to the definition of macro *name*.

Dependencies

```
main.o: main.cpp tools.hpp
tools.o: tools.cpp tools.hpp
```

Dependencies are a kind of degenerate rule.

- To build `main.o`, first “build” `main.cpp` and `tools.hpp`.
- To build `tools.o`, first “build” `tools.cpp` and `tools.hpp`.

But those dependencies are source files, so there is nothing to build. And where is the rule to build `main.o`?

What make does is compare the file modification dates on the target and on the dependencies in order to know if the target needs to be rebuilt.

Implicit rules

To build a target such as `main.o` for which there is no explicit rule, `make` uses an **implicit rule** that knows how to build any `.o` file from the corresponding `.cpp` file. In this case, the implicit rule invokes the `$(CXX)` compiler to produce output `main.o`. The compiler is called with the switches listed in `$(CXXFLAGS)`.

Integrated Development Environments

Graphical development tools: IDEs

Integrated Development Environments provide graphical tools to aid the programmer in many common tasks:

- Manage source files comprising a project;
- Display syntactic structure while editing;
- Search/replace over multiple files;
- Easy refactoring;
- Identifier completion;
- Display compiler error output in more readable form;
- Simplify edit-compile-run development cycle;

Recommended IDE's

Eclipse/CDT is a powerful, well-supported IDE that runs on many different platforms. Xcode is an Apple-proprietary IDE that only runs on Macs. Mac users may prefer it for its greater stability and even more features. I recommend either of these for serious C++ code development.

Geany is a lightweight IDE. It starts up much faster and is much more transparent in what it does. It should be more than adequate for this course.

Both Eclipse and Geany are installed on the Zoo, ready for your use.

The early part of this course can be perfectly well done in Emacs, so you don't have to learn Eclipse or Geany in order to get started.

Integrated Development Environment (e.g., Eclipse)

Advantages

- Supports notion of *project* — all files needed for an application.
- Provides graphical interface to all aspects of code development.
- Automatically creates `makefile`.
- Builds project with a mouse click or keyboard shortcut.
- Analyzes code as it is being written. Provides helpful feedback.
- Allows easy navigation among project components.
- Error comments are linked back to source code.

Integrated Development Environment (e.g., Eclipse)

Disadvantages

- Complicated to learn how to use — big learning curve.
- "Simple" things can become complicated for the non-expert (e.g., providing compiler flags) or making the font larger.
- Metadata can become inconsistent and difficult to repair.

Integrated Development Environment

If you use an IDE, before submitting your assignment, you should:

1. Copy your source code and test data files from the IDE to a separate `submit` directory *on the Zoo*.
2. Create a `Makefile` to build your project.
3. Test that everything works. Type `make` to make sure the project builds. Then run the resulting executable on your test suite to make sure it still does what you expect.

Submission Instructions

Submitting your assignments

1. Create a submission directory in your Zoo account named `ps1-netid123`, where you replace "ps1" with the current assignment number and "netid123" with your own net id.
2. Copy into it all the files you intend to submit.
3. Type `make` in that directory to make sure all needed files are present and your program builds and runs correctly.
4. Create required output files from your test runs.
5. Create a notes file that describes the submitted files.
6. Go up a level and create a gzipped tar file `ps1-netid123.tar.gz` using the command `tar -czvf ps1-netid123.tar.gz ps1-netid123`.
7. Submit the file `ps1-netid123.tar.gz` using Canvas.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 3
September 5, 2018

Insertion Sort Example

Design process: Insertion Sort

Here's a simple problem similar to what might be taught in a second programming class.

> *Write a C++ program to sort a file of numbers.*

This is hardly a specification. A few questions immediately come to mind:

- What file?
- What kind of numbers?
- What sorting algorithm should be used?
- Where does the output go?

A more refined specification

Here's a more detailed specification. The program should:

1. Prompt the user for the name of a file containing numbers.
2. The numbers are assumed to be floating point, one per line.
3. The numbers should be sorted using insertion sort.
4. The output should be written to standard output.

A first solution

03-InsertionSortMonolith satisfies the requirements.

Characteristics:

- It's monolithic – everything is in `main()`.
- It defines `BT` to be the type of number to be sorted. The definition uses a `typedef` statement.
- It uses dynamic storage to hold the list of numbers to be sorted.
- The macro `LENGTH` gives the maximum size list that it can handle. `#define` defines it to be 20.
- It proceeds in a logical step-by-step fashion through the entire solution process.

What is wrong with this?

This code violates many of the design principles I talked about in the first two lectures:

- Lack of isolation between the parts of the code that interact with the user, manage the dynamic storage, read the file, perform the sort, and print the results.
- It is not modular.
  - Variables used by the different parts are mixed together.
  - The storage management is intertwined with the other activities.
  - I/O and computation are mixed together.
- Reuse of the sorting algorithm is surprisingly difficult because of its entanglement with the other parts of the program.

A modular solution

03-InsertionC is a more modular solution that follows many OO-design principles, *even though it is written in C*.

- `main()` sequences the steps of the solution but delegates the implementation to functions defined in `databack.h`.
- `datapack.h` declares a `stuct DataPack` that brings together the variables needed to adequately represent the data to be processed.

A modular solution (cont.)

- `datapack.h` also declares three functions that make use of a `struct DataPack`:
  - `setup()` prompts the user for a file name, creates a `DataPack`, and initializes it with the data from the file.
  - `printData()` writes a `dataPack` to an output stream.
  - `sortData` sorts the data in a `dataPack`.
- `datapack.c` contains the implementations of these three functions.
- It also contains a *private* function `readData()` that does the actual user interaction for `setup()`. The `static` keyword in C restricts visibility of `readData()` to this one file.

C++ version

03-InsertionSortCpp is a solution written in C++ that uses many C++ features to achieve greater modularity than was possible in C.

It mirrors the file structure of the C version with the three files `main.cpp`, `datapack.hpp`, and `datapack.cpp`.

It achieves better modularity primarily by its use of **classes**. We give a whirlwind tour of classes in C++, which we will be covering in greater detail in the coming lectures.

Classes

Header file format

A `class` definition goes into a header file.

The file starts with **include guards**.

```
#ifndef DATAPACK_H
#define DATAPACK_H
// rest of header
#endif
```

or the more efficient but non-standard replacement:

```
#pragma once
// rest of header
```

Class declaration

Form of a simple class declaration.

```
class DataPack {
  private: // ---------------------------------------------------
    // data member declarations, like struct in C
    ...
    // private function methods
    ...
  public: // ----------------------------------------------------
    // constructor and destructor for the class
    DataPack() {...}
    ~DataPack() {...}
    }
    // ------------------------------------------------------------
    // public function methods
    ...
};
```

class DataPack

```
class DataPack {
...
};
```

defines a new class named `DataPack`.

By convention, class names are capitalized.

Note the *required* semicolon following the closing brace.

Class elements

- A class contains declarations and optionally definitions for *data members* and *function members* (or *methods*).
- `int n;` declares a data member of type `int`.
- `int size(){ return n; }` is a complete member function definition.
- `void sort();` declares a member function that must be defined elsewhere.
- By convention, member names begin with lower case letters and are written in camelCase.

Inline functions

- Methods defined inside a class are *inline* (e.g., `size()`).
- Inline functions are recompiled for every call.
- Inline avoids function call overhead but results in larger code size.
- `inline` keyword makes following function definition inline.
- Inline functions must be defined in the header (.hpp) file. Why?

Visibility

- The visibility of declared names can be controlled.
- `public:` declares that following names are visible outside of the class.
- `private:` restricts name visibility to this class.
- Public names define the interface to the class.
- Private names are for internal use, like local names in functions.

Constructor

A *constructor* is a special kind of method.

It name is the same as the class, and no return type is declared.

It is automatically called whenever a new class instance is created.

Its job is to initialize the raw data storage of the instance to become a valid representation of an initial data object.

In the `DataPack` example, `store` point to a block of storage with enough bytes to contain `max` items of type `BT`. The number of items currently in the store is kept in the data member `n`.

Constructor

```
DataPack(){
    n = 0;
    max = LENGTH;
    store = new BT[max]; cout << "Store allocated.\n";
    read();
}
```

`new` does the job of `malloc()` in C.

`cout` is name of standard output stream (like `stdout` in C).

`<<` is output operator.

`read()` is a private function to read data set from user.

Design question: Why is this a good idea?

Destructor

A *destructor* is a special kind of method.

It is automatically called whenever a class instance is about to be deallocated.

Its job is to perform any final processing of the data object such as returning any previously-allocated storage to the system.

In the `DataPack` example, the storage block pointed to by `store` is deallocated by the destructor.

Destructor

```
~DataPack(){
    delete[] store;
    cout << "Store deallocated.\n";
}
```

Name of the destructor is class name prefixed with `~`.

`delete` does the job of `free()` in C.

Empty square brackets `[]` are for deleting an array.

`dataPack.cpp`

Ordinary (non-inline) functions are defined in a separate *implementation file*.

Each defined function name must be prefixed with class name followed by `::` to identify which class's member function is being defined.

Example: `DataPack::read()` is the member function `read()` declared in class `DataPack`.

File I/O

C++ file I/O is described in Chapter 3 of *Exploring C++*. Please read it.

`ifstream infile( filename );` creates and opens an input stream `infile`.

The Boolean expression `!infile` is true if the file failed to open.

This works because of a built-in coercion from type `ifstream` to type `bool`. (More later on coercions.)

`read()` has access to the private parts of class `DataPack` and is responsible for maintaining their consistency.

main.cpp

As usual, the header file is included in each file that needs it:
`#include "datapack.hpp"`

`banner();` should be the first line of every program you write for this course. It helps debugging and identifies your output. (Remember to modify `tools.hpp` with your name as explained in Chapter 1 of textbook.)

Similarly, `bye();` should be the last line of your program before the return statement (if any).

The real work is done by the statements `DataPack theData;` which creates an instance of `DataPack` called `theData`, and `theData.sort();` which sorts `theData`. Everything else is just printout.

Manual compiling and linking

One-line version

```
g++ -O1 -g -Wall -std=c++17 -o isort main.cpp datapack.cpp tools.cpp
```

Separate compilation

```
g++ -c -O1 -g -Wall -std=c++17 -o datapack.o datapack.cpp
g++ -c -O1 -g -Wall -std=c++17 -o main.o main.cpp
g++ -c -O1 -g -Wall -std=c++17 -o tools.o tools.cpp
g++ -O1 -g -Wall -std=c++17 -o isort main.o datapack.o tools.o
```

Compiling and linking using `make`

The sample Makefile given in lecture 02 slide 28 is easily adapted for this project.

Compare it with the Makefile on the next slide.

Building InsertionSortCpp

```
#----------------------------------------------------------------
# Macro definitions
CXXFLAGS = -O1 -g -Wall -std=c++17
OBJ = main.o datapack.o tools.o
TARGET = isort
#----------------------------------------------------------------
# Rules
all: $(TARGET)
$(TARGET): $(OBJ)
	$(CXX) -o $@ $(OBJ)
clean:
	rm -f $(OBJ) $(TARGET)
#----------------------------------------------------------------
# Dependencies
datapack.o: datapack.cpp datapack.hpp tools.hpp
main.o: main.cpp datapack.hpp tools.hpp
tools.o: tools.cpp tools.hpp
```

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 4
September 10, 2018

C++ I/O

End of File and I/O Errors

C++ I/O

Streams

C++ I/O is done through **streams**.

Four standard streams are predefined:

- `cin` is the standard input stream.
- `cout` is the standard output stream.
- `cerr` is the standard output stream for errors.
- `clog` is the standard output stream for logging.

Data is read from or written to a stream using the input and output operators:

`>>` (for input).	Example: `cin >> x >> y;`
`<<` (for output).	Example: `cout << "x=" << x;`

Opening and closing streams

You can use streams to read and write files.

Some ways of opening a stream.

- `ifstream fin( "myfile.in" );` opens stream `fin` for reading. This implicitly invokes the constructor `ifstream( "myfile.in" )`.
- `ifstream fin;` creates an input stream not associated with a file. `fin.open( "myfile.in" );` attaches it to a file.

Can also specify open modes.

To test if `fin` failed to open correctly, write `if (!fin) {...}`.

To close, use `fin.close();`.

Reading data

Simple forms. Assume `fin` is an open input stream.

- `fin >> x >> y >> z;` reads three fields from `fin` into `x`, `y`, and `z`.
- The kind of input conversion depends on the types of the variables.
- No need for format or `&`.
- Standard input is called `cin`.
- Can read a line into buffer with `fin.get( buf, buflen );`. This function stops before the newline is read. To continue, one must move past the newline with a simple `fin.get(ch);` or `fin.ignore();`.

Writing data

Simple forms. Assume `fout` is an open output stream.

- `fout << x << y << z;` writes `x`, `y`, and `z` into `fout`.
- The kind of output conversion depends on the types of the variables or expressions..
- Standard output is called `cout`. Other predefined output streams are `cerr` and `clog`. They are usually initialized to standard output but can be redirected.
- Warning: The eclipse debug window does not obey the proper synchronization rules when displaying `cout` and `cerr`. Rather, the output lines are interleaved arbitrarily. In particular, a line written to `cerr` **after** a line written to `cout` can appear **before** in the output listing. This won't happen with a Linux terminal window.

Manipulators

Manipulators are objects that can be arguments of `>>` or `<<` but do not necessarily produce data.

Example: `cout << hex << x << y << dec << z << endl;`

- Prints `x` and `y` in hex and `z` in decimal.
- After printing `z`, a newline is printed and the output stream is flushed.

Manipulators are used in place of C formats to control input and output formatting and conversions.

Implementation of Manipulators

Manipulators are recognized by having a special function type, e.g, `std::ios_base& hex( std::ios_base& str );`.

The operators `>>` and `<<` have been predefined to recognize manipulators by their type and to take appropriate action when they are encountered.

Print methods in new classes

Each new class should have a `print()` function that writes out the object in human-readable form.

`print()` takes a stream reference as an argument that specifies which stream to write to.

The prototype for such a function should be:

```
ostream& print( ostream& out ) const;
```

If `sq` is an object of the new class, we can print `sq` by writing

```
sq.print(out);
```

Note that `const` prevents `print()` from modifying the object that it is printing.

Extending the I/O operators

While `sq.print()` allows us to print `sq`, we'd rather do it in the familiar way

`out << sq;`.

Fortunately, C++ allows one to extend the meaning of `<<` in this way. Here's how.

```
inline
ostream& operator<<( ostream& out, const Square& sq ) {
    return sq.print(out);
}
```

Since this function is inline, it should go in the header file for class `Square`.

Remarks on operator extensions

- Every definable operator has an associated function.
  The function for `<<` is `operator<<()`.
- Extending `<<` is simply a matter of defining the corresponding method for a new combination of parameters.
- In this case, we want to allow `out << sq`, where `out` has type `ostream&` and `sq` has type `const Square&`.
- The use of reference parameters prevents copying.
- The `const` is a promise that `operator<<` will not change `sq`.

Why << returns a stream reference

Both `print()` and `operator<<()` return a stream reference.

This allows compound constructs such as

```
out << "The square is:  " << sq << endl;
```

By left associativity of `<<`, this is the same as

```
((out << "The square is:  ") << sq) << endl;
```

Must it be inline?

If one wants `operator<<()` to be an ordinary function, the following changes are needed:

1. Declare the operator in header file `Square.hpp`:
   ```
   ostream& operator<<(ostream& out, const Square& sq);
   ```
2. Define the operator in code file `Square.cpp`:
   ```
   ostream& operator<<(ostream& out, const Square& sq) {
       return sq.print(out);
   }
   ```

End of File and I/O Errors

Status bits

I/O functions set status flags after each I/O operation.

- `badbit` means there was a read or write error on the file I/O.
- `failbit` means the data was not appropriate to the field, e.g., trying to read a non-numeric character into a numeric variable.
- `eofbit` means that the end of file has been reached.
- `goodbit` means that the above three bits are all off.

The whole state can be read with one call to `rdstate()`.

Status functions

Functions are also provided for testing useful combinations of status bits.

- `good()` returns true if the `good` bit is set.
- `bad()` returns true if the `bad` bit is set.

  This is *not* the same as `!good()`.
- `fail()` returns true if the `bad` bit or the `fail` bit is set.
- `eof()` returns true if the `eof` bit is set.

As in C, correct end of file and error checking require paying close attention to detail of exactly when these state bits are turned on. To continue after a bit has been set, must call `clear()` to clear it.

What eof means

Detecting and properly handling end of file is one of the most confusing things in C++.

The `eof` flag may or may not be on after the last byte of the file has been read and returned to the user.

The `eof` flag is turned on *when the stream attempts to read beyond the end of the file*.

To understand `eof` requires a thorough understanding of how stream input works.

When `eof` is turned on

A stream is a sequence of bytes. `>>` reads bytes until it has a complete representation of the object that it is trying to read.

Whether `eof` is turned on depends on whether or not the current input operation can complete based on the bytes read so far, *without looking ahead at the following byte*.

- If it needs the lookahead to detect completion and the bytes representing the data object go all the way to the end of the file, then it will try to read beyond the end of the file and will turn on the `eof` bit.
- If it doesn't need the lookahead, then it will stop reading, and the `eof` flag will remain off.

Reading an `int`

Consider what `cin >> x` does when reading the `int x`.

1. It first skips whitespace looking for the start of the number in the stream. It reads bytes one at a time until either there are no more left to read or a non-whitespace byte is read. If the first happens, no data is read into `x`, and both the `fail` and the `eof` flags are turned on (and the `good` flag is turned off).
2. If step 1 ended by finding a non-whitespace byte, then the stream checks if the character just read can begin an integer. The ones that can are `+, -, 0, 1, ..., 9`. If it is not one of these, the `fail` flag is set, the `eof` flag remains off, and nothing is stored into `x`.

Reading an `int` (cont.)

3. If an allowable number-starting character is found, then reading continues character by character until a character is read that can *not* be a part of the number currently being read, or the end of file is encountered so no more characters can be read.

   Reading then stops. If a stopping character was read, it is put back into the input buffer and the stream pretends that it was not read. If reading stopped because of an attempt to read past the end of the file, the `eof` flag is turned on.

   In either case, the characters read so far are converted to an `int`, stored into `x`, and the `fail` flag remains off. The `eof` flag is on iff reading was stopped by attempting to read past the end of the file.

Examples

The following examples show the remaining bytes in the file, where ␣ represents any whitespace character such as space or newline.

1. File contents: ␣123
   An attempt to read past the end of the file is made since otherwise one can't know that the number is 123 is complete. `good` and `fail` are off and `eof` is on.
2. File contents: ␣␣123␣
   `eof` will be off and the next byte to be read is the one following the 3 that stopped the reading. `good` is on and `fail` and `eof` are off.
3. File contents: ␣
   No number is present. Step 1 reads and discards the whitespace and attempts to read beyond the end of file. `good` is off and `fail` and `eof` are on.

Common file-reading mistakes

We now talk about the practical issue of how to write your code to correctly handle errors and end of file.

Two programming errors are common when reading data from a file:

- Failing to read the last number.
- Reading the last number twice.

Failing to read the last number

`good` is not always true after a successful read.

If the last number is *not* followed by whitespace, then after it is successfully read, `eof` is true and `good` is false. If one incorrectly assumes this means no data was read, the last number will not be processed.

Here's a naive program that illustrates this problem:

```
do {
   in >> x;
   if (!in.good()) break;
   cout << " " << x;
}
while (!in.eof());
cout << endl;
```

On input file containing 1␣2␣3, it will print ␣1␣2.

Reading the last number twice

eof is not always true after the last number is read.

If the last number *is* followed by whitespace, then after it is read, eof will still be false. If one incorrectly assumes it is okay to keep reading as long as eof is false, the last read attempt will fail and the input variable won't change.

Here's a naive program that illustrates this problem:

```
while (!in.eof()) {
   in >> x;
   cout << " " << x;
}
cout << endl;
```

On input file containing 1␣2␣3␣, it will print ␣1␣2␣3␣3.

How to read all numbers in a file

Here's a correct way to correctly read and process all of the numbers. Instead of printing them out, it adds them up in the register `s`.

```
int s=0;
int x;
do {
   in >> x;
   if (!in.fail()) s+=x;  // got good data
} while (in.good());
if (!in.eof()) throw Fatal("I/O error or bad data");
```

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 5
September 12, 2018

Functions and Methods

Call by value

Like C, C++ passes explicit parameters by value.

```
void f( int y ) { ... y=4; ... };
// Calling context
int x=3;
f(x);
```

- `x` and `y` are independent variables.
- `y` is created when `f` is called and destroyed when it returns.
- At the call, the *value* of `x` (=3) is used to initialize `y`.
- The assignment `y=4;` inside of `f` has no effect on `x`.

Call by pointer

Like C, pointer values (which I call **reference values**) are the things that can be stored in *pointer variables*.
Also like C, references values can be passed as arguments to functions with corresponding pointer parameters.

```
void g( int* p ) { ... (*p)=4; ... };
// Calling context
int x=3;
g(&x);
```

- `p` is created when `g` is called and destroyed when it returns.
- At the call, the *value* of `&x`, a reference value, is used to initialize `p`.
- The assignment `(*p)=4;` inside of `g` changes the value of `x`.

Call by reference

C++ has a new kind of parameter called a **reference parameter**.

```
void g( int& p ) { ... p=4; ... };
// Calling context
int x=3;
g(x);
```

- This does same thing as previous example; namely, the assignment `p=4` changes the value of `x`.
- Within the body of `g`, `p` is a synonym for `x`.
- For example, `&p` and `&x` are *identical* reference values.

I/O uses reference parameters

- The first argument to `<<` has type `ostream&`.
- `cout << x << y;` is same as `(cout << x) << y;`.
- `<<` returns a reference to its first argument, so this is also the same as

```
cout << x;
cout << y;
```

How should one choose the parameter type?

Parameters are used for two main purposes:

- To send data to a function.
- To receive data from a function.

Sending data to a function: call by value

For sending data to a function, *call by value* copies the data whereas *call by pointer or reference* copies only an address.

- If the data object is large, call by value is expensive of both time and space and should be avoided.
- If the data object is small (eg., an `int` or `double`), call by value is cheaper since it avoids the indirection of a reference.
- Call by value protects the caller's data from being inadvertantly changed.

Sending data to a function: call by reference or pointer

Call by reference or pointer allows the caller's data to be changed. Use `const` to protect the caller's data from inadvertant change.

Ex: `int f( const int& x )` or `int g( const int* xp )`.

Prefer call by reference to call by pointer for input parameters.

Ex: `f( 234 )` works but `g( &234 )` does not.

Reason: 234 is not a variable and hence can not be the target of a pointer.
(The reason `f( 234 )` *does* work is a bit subtle and will be explained later.)

Receiving data from a function

A parameter that is expected to be changed by the function is called an **output parameter**. (This is distinct from the function return value.)

Both call by reference and call by pointer work for output parameters.

Call by reference is generally preferred since it avoids the need for the caller to place an ampersand in front of the output variable.

Declaration: `int f( int& x )` or `int g( int* xp )`.

Call: `f( result )` or `g( &result )`.

The implicit argument

Every call to a class member function has an **implicit argument**. This is the object written before the dot in the function call.

```
class MyExample {
private:
   int count;    // data member
public:
   void advance(int n) { count += n; }
   ...
};
// Calling context
MyExample ex;
ex.advance(3);
```

Increments `ex.count` by 3.

`this` keyword

The implicit argument is passed by pointer.

It can be referenced directly from within a member function using the special keyword `this`.

In the call `ex.advance(3)`, the implicit argument is `ex`, and `this` acts like a pointer variable of type `MyExample*` that has been initialized to `&ex`.

Within the body of `advance()`, the variable name `count` and the expresssion `this->count` are synonymous. Both refer to the private data member `count`.

Derivation

Class relationships

Classes can relate to and collaborate with other classes in many ways.

We first explore **derivation**, where one class modifies and extends another.

What is derivation?

One class can be *derived* from another.

Syntax:

```
class Base {
public:
    int x;
    ...
};
class Deriv : public Base {
    int y;
    ...
};
```

`Base` is the base class; `Deriv` is the derived class.

`Deriv` inherits the members from `Base`.

Instances

A base class instance is contained in each derived class instance.

Similar to composition, except for inheritance.

Function members are also inherited.

Data and function members can be overridden in the derived class.

Derivation is a powerful tool for allowing variations to a design.

Some uses of derivation

Derivation has several uses.

- To allow a family of related classes to share common parts.
- To describe abstract interfaces à la Java.
- To allow generic methods with run-time dispatching.
- To provide a clean interface between existing, non-modifiable code and added user code.

Example: Parallelogram

```
class Parallelogram {
protected:           // allows access by children
    double base;     // length of base
    double side;     // length of side
    double angle;    // angle between base and side
public:              // public API
    Parallelogram() {}          // null default constructor
    Parallelogram(double b, double s, double a);
    double area() const;        // computes area
    double perimeter() const;   // computes perimeter
    ostream& print( ostream& out ) const;
};
```

Example: Rectangle

```
class Rectangle : public Parallelogram {
public:
   Rectangle( double b, double s ) {
      base = b;
      side = s;
      angle = pi/2.0; // assumes pi is defined elsewhere
   }
};
```

Derived class `Rectangle` inherits `area()`, `perimeter()`, and `print()` functions from `Parallelogram`.

Example: Square

```
class Square : public Rectangle {
public:
   Square( double b ) : Rectangle(b, b) {} // uses ctor
   bool inscribable( Square& s ) const {
      double diag = sqrt( 2.0 )*side;  // this diagonal
      return side <= s.side && diag >= s.side;
   }
   double area() const { return side*side; }
};
```

Derived class `Square` inherits the `perimeter()`, and `print()` methods from `Parallelogram` (via `Rectangle`).

It overrides the method `area()`.

It adds the method `inscribable()` that determines whether this square can be inscribed inside of its argument square `s`.

Notes on `Square`

Features of `Square`.

- The **ctor** `: Rectangle(b, b)` allows parameters to be supplied to the `Rectangle` constructor.
- The method `inscribable()` **extends** `Rectangle`, adding new functionality.
  It returns `true` if this square can be inscribed in square `s`.
- The function `area` overrides the less-efficient definition in `Parallelogram`.

Objects of Class Types

Structure of an object

A simple object is like a `struct` in C.
It consists of a block of storage large enough to contain all of its data members.

An object of a derived class contains an instance of the base class followed by the data members of the derived class.

Example:

```
class Deriv :  Base { ... };
Deriv myObj;
```

Then "inside" of `myObj` is a `Base`-instance!

Example object of a derived class

The declaration `Base bObj` creates a variable of type `Base` and storage size large enough to contain all of `Base`'s data members (plus perhaps some padding).

`bObj`: | `int x;` |

The declaration `Deriv dObj` creates a variable of type `Deriv` and storage size large enough to contain all of `Base`'s data members plus all of `Deriv`'s data members.

`dObj`: | | `int x;` | `int y;` |

The inner box denotes a `Base`-instance.

Referencing a composed object

Contrast the previous example to

```
class Deriv { Base bObj; ...};
Deriv dObj;
```

Here `Deriv` composes `Base`.

The variable `x` from the embedded `Base` object can be referenced using `bObj.x` .

Referencing a base object

How do we reference the base object embedded in a derived class?

Example:

```
class Base { public: int x; int y; ...};
class Deriv :  Base { int y; ...};
Deriv dObj;
```

- The data members of `Base` can be referenced directly by name.
  `x` refers to data member `x` in class `Base`.
  `y` refers to data member `y` in class `Deriv`.
  `Base::y` refers to data member `y` in class `Base`.
- `this` points to the whole object.
  Its type is `Deriv*`.
  It can be coerced to type `Base*`.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 6
September 17, 2018

Construction, Initialization, and Destruction

Reference Types

Construction, Initialization, and Destruction

Initializing an object

Whenever a class object is created, one of its constructors is called.

This applies not only to the "outer" object but also to all of its embedded objects.

If not specified otherwise, the default constructor is called, if defined. This is the one that takes no arguments.
Example: `MyClass mc;` calls default constructor `mc`.

If you do not define any constructors, then the default constructor is defined automatically to be the null constructor.

Which constructor gets used?

A class can have several constructor methods, differing from each other in the number and types of arguments.

When an object is created, the constructor called is the one matching the user-specified arguments.

For example, suppose the user declares two `Parallelogram` objects:

```
Parallelogram tempShape;
Parallelogram yellowShape( 5, 5, 30 );
```

`tempShape` is initialized by calling the null constructor.
`yellowShape` is initialized by calling `Parallelogram(5, 5, 30)`.

Construction rules for a simple class

The rule for constructing an object of a simple class is:

1. Call the constructor/initializer for each data member, in sequence.
2. Call the constructor for the class.

Construction rules for a derived class

The rule for constructing an object of a derived class is:

1. Call the constructor for the base class (which recursively calls the other constructors needed to completely initialize the base class object.)
2. Call the constructor/initializer for each data member of the derived class, in sequence.
3. Call the constructor for the derived class.

Destruction rules

When an object is deleted, the destructors are called in the opposite order before the storage allocated to the object is released back to the system.

The rule for an object of a derived class is:

1. Call the destructor for the dervied class.
2. Call the destructor for each data member of the derived class in reverse sequence.
3. Call the destructor for the base class.

Rules for a simple class are the same except that step 3 is omitted.

Constructor ctors

Ctors (short for constructor/initializors) allow one to supply parameters to implicitly-called constructors.

Example:

```
class Deriv : Base {
  Deriv( int n ) : Base(n) {};
      // Calls Base constructor with argument n
};
```

Initialization ctors

Ctors also can be used to initialze primitive (non-class) variables.

Example:

```
class  Deriv {
  int x;
  const int y;
  Deriv( int n ) : x(n), y(n+1) {}; //Initializes x and y
};
```

Multiple ctors are separated by commas.

Ctors present must be in the same order as the construction takes place – base class ctor first, then data member ctors in the same order as their declarations in the class.

Initialization not same as assignment

Previous example using ctors is not the same as writing

```
Deriv( int n ) { y=n+1; x=n; };
```

- The order of initialization differs.
- `const` variables can be initialized but not assgined to.
- Initialization uses the constructor (for class objects).
- Initialization from another instance of the same type uses the copy constructor.

Special member functions

A class has six special member functions. These are special because they are defined automatically if the programmer does not redefine them. They are distinguished by their prototypes.

Name	Prototype
Default constructor	`MyClass();`
Destructor	`~MyClass();`
Copy constructor	`MyClass( const MyClass& other );`
Move constructor	`MyClass( MyClass&& other );`
Copy assignment	`MyClass& operator=( const T& other );`
Move assignment	`MyClass& operator=( T&& other );`

Special function automatic definitions

Name	**Automatic Definition**
Default constructor	Null constructor does nothing;
Destructor	Function that does nothing
Copy constructor	Does a shallow copy from its argument
Move constructor	(later)
Copy assignment	Does a shallow copy from rhs to lhs
Move assignment	(later)

Copy and assignment have the same default semantics but can be redefined to behave differently.

Deletion

Some of the automatic definitions are omitted if certain special functions are defined by the user.

For example, if you define a constuctor with arguments, then the default constructor is automatically deleted.

You can explicitly remove any automatically-created special function by using `=delete` in place of a definition.

Example: To remove the copy constructor for `MyClass`, write
`MyClass( const MyClass& ) = delete;`

Restoration of automatically deleted definition

If a default definition for a special function is automatically deleted, it can be brought back using `=default` in place of a definition.

For example, if you define a constuctor with arguments, then the default constructor is automatically deleted.

To bring it back, you can write `MyClass() = default;`.

Copy constructors

- A copy constructor is automatically defined for each new class `MyClass` and has prototype `MyClass(const MyClass&)`. It initializes a newly created `MyClass` object by making a shallow copy of its argument.
- Copy constructors are used for call-by-value parameters.
- Assignment uses `operator=()`, which by default copies the data members but does not call the copy constructor.
- The results of the implicitly-defined assignment and copy constructors are the same, but they can be redefined to be different.

Move constructors

C++11 introduced a move constructor. Its purpose is to allow an object to be safely moved from one variable to another while avoiding the "double delete" problem.

We'll return to this interesting topic later, after we've looked more closely at dynamic extensions.

Reference Types

Reference types

Recall: Given `int x`, two types are associated with `x`: an L-value (the reference to `x`) and an R-value (the type of its values).

C++ exposes this distinction through *reference* types and declarators.

A *reference type* is any type `T` followed by `&`, i.e., `T&`.

A reference type is the internal type of an L-value.

Example: Given `int x`, the name `x` is bound to an L-value of type `int&`, whereas the values stored in `x` have type `int`

This generalizes to arbitrary types `T`: If an L-value stores values of type `T`, then the type of the L-value is `T&`.

Reference declarators

The syntax `T&` can be used to declare names, but its meaning is not what one might expect.

```
int x = 3;     // Ordinary int variable
int& y = x;    // y is an alias for x
y = 4;         // Now x == 4.
```

The declaration must include an initializer.

The meaning of `int& y = x;` is that `y` becomes a name for the L-value `x`.

Since `x` is simply the name of an L-value, the effect is to make `y` an alias for `x`.

For this to work, the L-value type (`int&`) of `x` must match the type declarator (`int&`) for `y`, as above.

Use of named references

Named references can be used just like any other variable.

One application is to give names to otherwise unnamed objects.

```
int axis[101];              // values along a graph axis
int& first = axis[0]   ;    // give name to first element
int& last = axis[100];      // give name to last element
first = -50;
last = 50;

// use p to scan through the array
int* p;
for (p=&first; p!=&last; p++) {...}
```

Reference parameters

References are mainly useful for function parameters and return values.

When used to declare a function parameter, they provide call-by-reference semantics.

```
int f( int& x ){...}
```

Within the body of `f`, `x` is an alias for the actual parameter, which must be the L-value of an `int` location.

Reference return values

Functions can also return references.

```
int& g( bool flag, int& x, int& y ) {
    if (flag) return x;
    return y;
}
...
g(x<y, x, y) = x + y;
```

This code returns a reference to the smaller of `x` and `y` and then sets that variable to their sum.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 7
September 19, 2018

Reference Types (cont.)

Reference Types (cont.)

Custom subscripting

Suppose you would like to use 1-based arrays instead of C++'s 0-based arrays.

We can define our own subscript function so that `sub(a, k)` returns the L-value of array element `a[k-1]`.

`sub(a,k)` can be used on either the left or right side of an assignment statement, just like the built-in subscript operator.

```
int& sub(int a[], int k) { return a[k-1]; }
...
int mytab[20];
for (k=1; k<=20; k++)
    sub(mytab, k) = k;
```

Constant references

Constant reference types allow the naming of pure R-values.

```
const double& pi = 3.14159265358979323846264338327950;
```

Actually, this is little different from

```
const double pi = 3.14159265358979323846264338327950;
```

In both cases, the pure R-value is placed in a read-only object, and `pi` is bound to its L-value.

A review of definitions

- An **object** is a block of memory into which data can be stored along with a type.
- The **type** of an object tells the storage size and interpretation of its contents.
- The **R-value** of an object is the sequence of bytes stored in it.
- The **L-value** of an object is a unique label for the object. It is often represented by a machine address.
- A **reference** is an L-value along with its type.
- An object might or might not have a **name**. If it does, the name is **bound** to a reference.

LHS and RHS contexts

- The meaning of a name or reference depends on the context in which it appears.
- The right hand side of an assignment statement is said to be **RHS context**. A name appearing there evaluates to the R-value of the object that it references.
- The left hand side of an assignment statement is said to be **LHS context**. A name appearing there evaluates to the L-value of the object that it references.

Example

`int x = 3` creates an object on the stack of type `int`, stores the number 3 in it, and gives it the name "`x`".

Let `0x1234` be the address of the newly-created object `x`.

- The L-value of `x` is `0x1234`;
- The R-value of `x` is `3`;
- `x` itself names the reference `(0x1234, int)`.

In the expression `y = x+1`, the name `x` appears in RHS context.
Its R-value, `3`, is fetched from `x` and used by the `+` operator.
The name `y` appears in LHS context.
Its L-value is where the result of `x+1` is stored.

Pointers

A **pointer** is a special kind of R-value that embeds a reference.

The prefix operator `*`, applied to a pointer, returns the reference embedded in the pointer. This operation is called **following the pointer**.

A pointer that embeds a reference of type `T` is said to have type `T*`.

If `x` is a reference of type `T`, then the prefix operator `&` can be applied to `x` to produce a pointer to `x`.

The type of `&x` is `T*`. Thus, `*&x` is an alias for `x`.

Pointer objects

- A **pointer object** of type `T*` is an object that can store pointers of type `T*` as its R-values.
- The star operator `*p` applied to a pointer object `p` first fetches the R-value of `p` which is a pointer. It then follows that pointer and returns its embedded reference.
- This returned reference can be used like any other object. For example, if `p` has type `int*`, then `(*p) = 17` stores 17 into the reference returned by `*p`, which will have type `int`.

Examples Presented in Class

Several examples were presented in class on the blackboard.

Hand-drawn pictures used boxes to represent objects, hex numbers to represent L-values, numbers inside boxes to represent primitive R-values, and arrows starting inside one box and pointing to another to represent pointers.

Anyone who missed class is encouraged to borrow class notes from someone who attended.

Comparison of reference and pointer

- A reference (L-value) is the result of following a pointer.
- A pointer is only followed when explicitly requested (by `*` or `->`).
- A reference name is bound when it is created. Pointer objects can be initialized at any time (unless declared to be `const`).
- Once a reference is bound to an object, it cannot be changed to refer to another object. Pointer objects can be be assigned a different pointer at any time (unless declared to be `const`).
- A reference is always associated with a fixed piece of storage. By way of contrast, a pointer object can contain the special value `nullptr`, which is a special pointer that can be compared for equality but not be followed.

Concept summary

Concept	Meaning
Object	A block of memory and its contents.
L-value	The machine address of an object.
R-value	The value stored in an object.
Pointer	An R-value consisting of a machine address.
Pointer object	An object into which a pointer can be stored.
Reference	A typed L-value.
Identifier	A name which is bound to a reference.

Type summary

Let `T` be any type.

Concept	Type	Meaning
Object	`T`	L-value has type `T&`, R-value has type `T`.
L-value	`T&`	The object at its address has type `T`.
R-value	`T`	The type of the data value is `T`.
Pointer object	`T*`	L-value has type `T*&`, R-value has type `T*`.
L-value of ptr obj	`T*&`	The object at its address has type `T*`.
Pointer R-value	`T*`	The type of the data value is `T*`.

Declaration syntax

`T x;`	Binds `x` to the L-value of a new object of type `T`.
`T& x=y;`	Binds `x` to the L-value of `y`, which has type `T&`.
`T* x = new T;`	Binds `x` to the L-value of a new pointer object `x` of type `T*`, creates a dynamically-allocated object of type `T`, and stores a pointer to it in `x`.
`T* y;`	Binds `y` to a new uninitialized object of type `T*`.

Storing a list of objects in a data member

A common problem is to store a list of objects of some type `T` as a data member `li` in a class `MyClass`.

Here are six ways it can be done:

1. `T li[100];` — `li` is *composed* in `MyClass`.
2. `T* li[100];` — `li` is *composed* in `MyClass`. Constructor does loop to store `new T` in each array slot.
3. `T* li;` — Constructor does `li = new T[100];`.
4. `T** li;` — Constructor does `li = new T*[100];` then does loop to store `new T` in each array slot.
5. `vector<T> li;` — Uses Standard `vector` class. `T` must be copiable.
6. `vector<T*> li;` — Constructor does loop to store `new T` into each vector slot.

How to access

Here's how to acces element 3 in each case:

1. `T li[100];` `li[3]`.
2. `T* li[100];` `*li[3]`.
3. `T* li;` `li[3]`.
4. `T** li;` `*li[3]`.
5. `vector<T> li;` `li[3]`.
6. `vector<T*> li;` `*li[3]`.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 8
September 24, 2018

Etudes in Coding

Overview

Software construction is much like other activities that combine design with skills.

Piano students practice scales and études as well as learning to play Beethoven piano sonatas.

Ballet dancers do barre exercises to acquire the skills needed to dance Nutcracker.

Authors learn good writing style by having others criticize their own work.

Today I present some examples of programs and try to point out the design decisions that impact the cleanliness and robustness of the result.

Problem Set 1 Design Issues

```
// Solution by Michael J. Fischer
// Calculate a user's age
void run() {
    string first;
    string last;
    int birthYear;
    int age;

    // Get current year
    const time_t now = time( nullptr );         // get current time
    struct tm* today = localtime( &now );       // break into parts yr-mon-day
    const int thisYear = 1900 + today->tm_year; // tm_year counts years from 1900

    cout << "Please enter your first name: ";
    cin >> first;
    if (!cin.good()) fatal("Error reading first name");

    cout << "Please enter your last name: ";
    cin >> last;
    if (!cin.good()) fatal("Error reading last name");

    cout << "Please enter the year of your birth: ";
    cin >> birthYear;
    if (!cin.good()) fatal("Error reading age");

    age = thisYear - birthYear;
    cout << first << " " << last << " becomes " << age << " years old in "
         << thisYear << "." << endl;
}
```

Comments on my code

Good points:

- Logical progression towards solution: get year, get first name, get last name, get birth year, compute age, print results.
- Most obscure part of getting current year is commented.
- Identifiers are compromise between length and clarity.
- All I/O errors are detected, reported, and handled as required.

Drawbacks:

- Code is monolithic.
- User-interaction is intermixed with computation.
- Variables related to user (`first, last, birthYear, age`) are not separated from intermediate variables (`now, today, thisYear`).
- General computation is not isolated from input-specific code.

A student solution, function isgood()

```
// ---------------------------------------------------------------------------
// Function to check for input errors and then concatenate first and last name
// string inputs.
void isgood(string *name, string *temp)
{
    cin >> *temp;
    if (cin.good()) {
        *name = *name + *temp;
    }
    else {
        fatal("Invalid input.");
    }
}
```

Comments on `isgood()`

Good points:

- Clear separation from surrounding code.
- Clear statement of purpose, but incomplete.
- Uses `cin.good()` for error checking as required.

Drawbacks:

- Statement of purpose omits mention of string read.
- Function name suggests only the checking part.
- A check-only founction should be `const` and return a `bool`.
- The actions to take with a successful or unsuccessful read should not be the concern of the checking function.
- `name` should not be a parameter.
- Output parameter `temp` should be of reference type `string&`.

A student solution, function calctime()

```
// ---------------------------------------------------------------------------
// Function to check for input errors and then calculate both the current year
// and the age of the user using time() and localtime().
void calctime(int *age, int *year)
{
    int birth;
    cin >> birth;
    if (cin.good()) {
        time_t current;
        struct tm * localhold;

        time(&current);
        localhold = localtime(&current);

        *year = 1900 + localhold->tm_year;
        *age = *year - birth;
    }
    else {
        fatal("Invalid input.");
    }
}
```

Comments on `calctime()`

Similar coments to `isgood()`.

Main drawback is that user interaction, data reading, error checking, and time calculations are carried out by the same function.

When we get to classes,`age` and `year` would be data members of the class containing `calctime()`, and `calctime()` would need no parameters.

Minor formatting problem: Left bracket { should be at end of `isgood` line, not on a line by itself. Applies to `isgood()` as well.

A student solution, function run()

```
// ------------------------------------------------------------------------------
// Run function that prints out user prompts and calls subsidiary functions for
// processing submitted inputs.
void run() {
    string name;
    string temp;

    cout << "Please enter your first name: ";
    isgood(&name, &temp);
    name = name + " ";              // adds a space between first and last name
    cout << "Please enter your last name: ";
    isgood(&name, &temp);

    int age;
    int year;

    cout << "Please enter the year of your birth: ";
    calctime(&age, &year);

    cout << name << " becomes " << age << " years old in " << year << ".\n";
}
```

Comments on `run()`

Good points:

- Correctly formatted function definition.
- Checks both first name and last name for read errors.
- Checking code is not replicated.
- Consistent top-level structure for handling names and birth year.

Drawbacks:

- No need to use expensive string concatenation. `name` is unnecessary. Better to have separate `first` and `last` string variables.

Brackets Example

Code demo

The 08-Brackets demo contains three interesting classes and illustrates the use of constructors, destructors, and dynamic memory management as well as a number of newer C++ features.

It is based on the example in section 4.5 of "Exploring C++", but there are several significant modifications to the code.

Many of the changes use features of c++17 and would not work under the older standard. Others reflect different design philosophies.

We briefly summarize below some of the features of the demo.

The problem

The problem is to check a file to see if the brackets match and are properly nested.

For example, `([]())` is okay, but `([)]` is not, nor is `(()))` or `[[[`.

A bracket matching algorithm

Rules for bracket matching:

1. Each left bracket is pushed onto the stack.
2. An attempt is made to match each right bracket with the top character on the stack.
3. The attempt fails if
   - The stack is empty, or
   - The top character is a different type of bracket (e.g., round instead of square).
4. If the match fails, an error comment is printed, the mismatched characters are discarded, and processing continues with the next character.
5. At end-of-file, the stack should be empty, for any remaining characters on the stack are unmatched left brackets.

Program design

The program is organized into four modules.

1. Class `Token` wraps a single character. It contains functions for determining which characters are brackets, and for each bracket, its "sense" (left or right), and its "type" (round, square, curly, or angle).
2. Class `Stack` implements a general-purpose growable stack of objects of copyable type `T`. In this case, `T` is typedef'ed to `Token`.
3. Class `Brackets` implements the matching algorithm. It reads the file and carries out the matching algorithm.
4. `main.cpp` contains the main program. It processes the command line, opens the file, and invokes the bracket checker.

Token class

Major points:

1. `enum` is used to encode the bracket type (round, square, etc.) and the sense of the bracket (left, right).
2. The two `enum` types are defined inside of class `Token` and are private.
3. `ch` is the character representing the bracket, used for printing.
4. `classify()` is a private function.
5. The definitions of `print()` and `operator<<` follow our usual paradigms.

Token class (cont.)

6. The `Token` constructor uses a ctor to initialize data member `ch`. This overrides the **default member initializer** present in the declaration of `ch`. The constructor calls `classify()` to initialize the other data members.
7. In the ctor `:ch(ch)` , the first `ch` refers to the data member and the second refers to the constructor argument.
8. In the textbook version of `Token`, the static object `brackets` is *local* to `classify()`. It is now a static *class object*, initialized in `token.cpp`.

Token design questions

1. The textbook version of `Token` uses getters to return `type` and `sense`. `getType()` was used to test if a newly-read character was a bracket, and it was also used to see if a left bracket and right bracket were the same type.
   Why were they needed?
2. The new version of `Token` replaces `getType()` with boolean functions `isBracket()` and `sameTypeAs()` functions. Similarly, `getSense()` was replaced by boolean function `isLeft()`.

   With these changes, enum `BracketType` and `TokenSense` are no longer needed outside of `Token` and hence are now private.

   What are the pros and cons of this design decision?

Token design questions (cont.)

3. Both the old and new versions of the program work whether or not `brackets` is `static`.
   - Is static a better choice here?
   - Why or why not?
   - Does your answer depend on whether the object is local (old code) or class (new code)?

Stack class

Major points:

1. `T` is the element type of the stack. This code implements a stack of `Token`. (See `typedef` declaration.)
2. Storage for stack is dynamically allocated in the constructor using `new[]` and deleted in the destructor using `delete[]`.
3. The copy constructor and assignment operator have been deleted to avoid "double delete" problens with the dynamic extension.
4. The square brackets are needed for both `new` and `delete` since the stack is an array.
5. `delete[]` calls the destructor of each `Token` on the stack. Okay here because the token destructor is null.

Stack class (cont.)

6. `push()` grows stack by creating a new stack of twice the size, copying the old stack into the new, and deleting the old stack. This results in linear time for the stack operations.
7. If `push()` only grew the stack one slot at a time, the time would grow quadratically.

Stack design questions

1. Should `pop()` return a value?
2. Why does stack have a `name` field?
3. `size()` isn't used. Should it be eliminated?
4. `Stack::print()` formerly declared `p` and `pend` at the top. Now they are declared just before the loop that uses them. Is this better, and why?
5. Could they be declared in the loop? What difference would it make?

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 9
September 26, 2018

These abbreviated notes summarize lecture 9 given on September 26 but do not by any means fully capture what was presented.

Following Specifications

Why follow instructions?

A reasonable question is, "Why should I follow instructions when I know a different or better way of accomplishing the "same thing"?

1. Programming is about producing code that fully satisfies design requirements.
2. If you don't like the requirements, it's reasonable to question them but not simply to ignore them.
3. For this course, the problem requirements also have a pedagogical purpose. When I say, for example, that a goal of the assignment is to learn how to use the C time functions from within C++, I mean exactly that. I'm not asking you to just figure out some way of determining the current year.
4. The ability to understand and follow instructions is a sign of maturity and professionalism that will help you in your career.

Bytes and Characters

History of ASCII

We had a long discussion of the history of character encodings, starting from 7-bit ASCII as used on early teletype machines up to current-day unicode.

Originally, the only characters that could be encoded on a computer were the ones that appeared on an English-language typewriter. There are so few such charcters that they can be encoded in a single 8-bit byte.

At the time C was created, ASCII characters were all that were important to be able to read and write. Hence, type `char` became the name of a single-byte storage unit that could be used to represent a character (but could be used for other purposes as well).

Unicode

Unicode is a standard that assigns a unique numerical code to every letter and symbol in every language on earth. There are so many characters that the unicode encoding needs 32 bits.

These 32-bit quantities are usually themselves represented as sequences of one or more shorter storage units.

The commonly-used utf-8 encoding is a way of representing every unicode character by a sequence of one or more 8-bit bytes.

C/C++ works directly with bytes, not characters. A function like `in.get(ch)` reads a byte into `ch`, not a full character.

Note: The utf-8 encoding of every ASCII character is a single byte whose value is the same as its ASCII code.

Overview of PS3

Think-a-Dot

I gave an overview of the *Think-a-Dot* game. Everything I said is contained in the PS3 assignment and in some of the references cited there.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 10
October 1, 2018

Brackets Example (continued from lecture 8)
- Stack class
- Brackets class
- Main file

Storage Management

Brackets Example (continued from lecture 8)

Stack class

Major points:

1. `T` is the element type of the stack. This code implements a stack of `Token`. (See `typedef` declaration.)
2. Storage for stack is dynamically allocated in the constructor using `new[]` and deleted in the destructor using `delete[]`.
3. The copy constructor and assignment operator have been deleted to avoid "double delete" problems with the dynamic extension.
4. The square brackets are needed for both `new` and `delete` since the stack is an array.
5. `delete[]` calls the destructor of each `Token` on the stack. Okay here because the token destructor is null.

Stack class (cont.)

6. `push()` grows stack by creating a new stack of twice the size, copying the old stack into the new, and deleting the old stack. This results in linear time for the stack operations.
7. If `push()` only grew the stack one slot at a time, the time would grow quadratically.

Stack design questions

1. Should `pop()` return a value?
2. Why does stack have a `name` field?
3. `size()` isn't used. Should it be eliminated?
4. `Stack::print()` formerly declared `p` and `pend` at the top. Now they are declared just before the loop that uses them. Is this better, and why?
5. Could they be declared in the loop? What difference would it make?

Brackets class

1. Data member `stk` is dynamically allocated in the constructor and deleted in the destructor. It is an object, not an array, and does *not* use the `[]`-forms of `new` and `delete`.
2. The type of `stk` has changed from `Stack*` to `Stack`. We can now print the stack by writing `cout << stk`. Formerly, we wrote `stk->print(cout)`.
3. `in.get(ch)` reads the next character without skipping whitespace. There are other ways to do this as well.
4. If read is `!in.good()`, we `break` from the loop and do further tests to find the cause.
5. Old functions `analyze()` and `mismatch()` have been replaced by `checkFile()` and `checkChar()`. This largely separates the file I/O from the bracket-checking logic.

Brackets design questions

- What are the pros and cons of `stk` having type `Stack&` rather than `Stack*`?
- The old `mismatch()` uses the `eofile` argument to distinguish two different cases.

```
void Brackets::
mismatch( const char* msg, Token tok, bool eofile ) {
  if (eofile) cout <<"\nMismatch at end of file: " <<msg <<endl;
  else        cout <<"\nMismatch on line " <<lineno <<" : " <<msg <<endl;

  stk->print( cout );   // print stack contents
  if (!eofile)          // print current token, if any
      cout <<"The current mismatching bracket is " << tok;
  fatal("\n");          // Call exit.
}
```

Is this a good design?

Main file

1. `main()` follows our usual pattern, except that it passes `argc` and `argv` on to the function `run()`, which handles the command line arguments.
2. `run()` opens the input file and passes the stream `in` to `analyze()`.
3. The istream `in` will not be closed if an error is thrown (except for the automatic cleanup that happens when a program exits). How might we fix the program?
4. Question: Which is better, to pass the file name or an open stream? Why?

Storage Management

Objects and storage

Objects have several properties:

- A **name**. This is one way to access the object.
- A **type**. This determines the size and encoding of the allowable **data values**.
- A **storage block**. This is a block of memory big enough to hold any legal value of the specified type.
- A **lifetime**. This is the time span between an object's creation and its demise. Data left behind in an object's storage block after it has died is unpredictable and shouldn't be used.
- A **storage class**. This determines the lifetime of the object, where the storage block is located in memory, and how it is managed.

Name

An object may have one or more names, or none at all!

Not all names are created equal. A name may exist but not be visible in all contexts.

- It is not visible from outside of the block in which it is defined.
- For a class data member, the name's visibility may be restricted, e.g., by the `private` keyword.
- An object may have more than one name. This is called **aliasing**.
- An object may have no name at all. Such an object is called **anonymous**. It can only be accessed via a pointer or subscript.

Type of a storage object

Declaration: `int n = 123;`

This declares an object of type `int`, name `n`, and an `int`-sized storage block, which will be initialized to 123. It's lifetime begins when the declaration is executed and ends on exit from the enclosing block. The storage class is `auto` (stack).

The unary operator `sizeof` returns the storage size (in bytes).

`sizeof` can take either an expression or a parentheses-enclosed type name, e.g., `sizeof n` or `sizeof(int)`.

In case of an expression, the size of the result type is returned, e.g., `sizeof (n+2.5)` returns 8, which is the size of a `double` on my machine.

Storage block

Every object is represented by a block of storage in memory.

This memory has an internal **machine address**, which is not normally visible to the programmer.

The size of the storage block is determined by the type of the object.

Connecting names to objects

A name can be given to an anonymous object at a later time by using a **reference** type.

```
#include <iostream>
using namespace std;
int main() {
  int* p;
  p = new int;  // Creates an anonymous int object
  *p = 3;       // Store 3 into the anonymous object
  cout << *p << endl;
  int& x = *p;  // Give object *p the name x
  x = 4;
  cout << *p << " " << x << endl;
}
/* Output
3
4 4
*/
```

Lifetime

Each object has a **lifetime**.

The lifetime begins when the object is **created** or **allocated**.

The lifetime ends when the object is **deleted** or **deallocated**.

Storage class

C++ supports three different storage classes.

1. `auto` objects are created by variable and parameter declarations. (This is the default.)
   Their visibility and lifetime is restricted to the block in which they are declared.
   The are deleted when control finally exits the block (as opposed to temporarily leaving via a function call).
2. `new` creates anonymous *dynamic* objects. They exist until explicitly destroyed by `delete` or the program terminates.
3. `static` objects are created and initialized at load time and exist until the program terminates.

Dynamic extensions

Recall that objects have a fixed size determined solely by the object type.

A variable-sized "object" is modeled in C++ by an object with a **dynamic extension**. This object has a pointer (or reference) to a dynamically allocated object (generally an array) of the desired size.

Example from stack.hpp.

```
class Stack {
private:
  int max = INIT_DEPTH; // Number of slots in stack.
  int top = 0;          // Stack cursor.
  T* s = new T[max];    // Pointer to stack base.
  string name;          // Print name of this stack.
  ...
```

Copying

A source object can be copied to a target object *of the same type*.

A **shallow copy** copies each source data member to the corresponding target data member. By default, this is done by performing a byte-wise copy of the source object's storage block to the target object's storage block, overwriting its previous contents.

For objects with dynamic extensions, the *pointer* to the extension gets copied, not the extension itself. This causes the target to end up sharing the extension with the source, and the target's previous extension becomes **inaccessible**. This results in **aliasing**—multiple pointers referring to the same object, which can cause a **memory leak**.

A **deep copy** recursively copying the extensions as well.

The double-delete problem

An object with dynamic extension typically uses `new` in the constructor and `delete` in the destructor to create and free the object.

When a shallow copy results in two objects sharing the same extension, then attempts will be made to delete the extension when each of the two copies of the object are deleted or go out of scope.

The first delete will succeed; the second will fail since the same object cannot be deleted twice.

This is called the **double delete** problem and is a major source of memory management errors in C++.

Takeaway: Don't copy objects with dynamic extensions.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 11
October 3, 2018

Copying and Assignment

When does copying occur?

C++ has two operators defined by default that make copies:

1. The assignment statement.
2. The copy constructor.

The symbol = means assignment when used in a **statement**, and it invokes the copy constructor when used as an **initializer**.
Call-by-value argument passing also uses the copy constructor.

Assignment modifies an existing object;

The copy constructor initializes a newly-allocated object.

Assignment

The **assignment** operator `=` is implicitly defined for all types. The assignment `b=a` modifies an already-existing object `b` as follows:

- If `a` and `b` are primitive types, the storage object `a` is copied to the storage object `b` (after performing any implicit conversions such as converting a `short int` to an `int`). In the case of pointer types, this results in `a` and `b` pointing to the same block of memory.
- If `a` and `b` are objects, then each data member of `a` is recursively assigned to the corresponding data member of `b`, using the assignment operator defined for the data member's type.

Copy constructor

The **copy constructor** is implicitly defined for all types. Like any constructor, it can be used to initialize a newly-allocated object.

- Call-by-value uses the copy constructor to initialize a function parameter from the actual argument.
- The copy constructor can also be used to initialize a newly-created object.

The implicit copy constructor uses shallow copy, so any use of it on an object with dynamic extension leads to the double delete problem.

Redefining assignment and the copy constructor

You can override the implicit assignment operator for a class `T` by defining the function with signature `T& operator=(const T&);`.

You can override the implicit the copy constructor by defining the function with signature `T(const T&)`.

If an implicit definition has been automatically deleted but you want it, use `=default`.

If an implicit definition has been automatically created but you don't want it, use `=delete`.

If you don't intend to use the copy assignment or constructor, deleting them prevents their accidental use.

Custody of Objects

Copying and Moving

One of the goals of C++ is to make user-defined objects look as much like primitive objects as possible.

In particular, they can reside in static storage, on the stack, or in the heap, they can be passed to and returned from functions, and they can be initialized and assigned to.

With primitive types, initialization, assignment, call-by-value parameters and function return values are all implemented by a simple copy of the primitive value.

The same is done with objects, but shallow copy is used by default.

This can lead to problems with large objects (cost) and with objects having dynamic extensions (double-delete problem) discussed above.

Custody

We say that a function or class has **custody** of a dynamically-allocated object if it is responsible for eventually deleting the object.

A simple strategy for managing a dynamic extension in a class is for the constructor to create the extension using `new` and for the destructor to free it using `delete`.

In this case, we say that custody remains in the class.

Transfer of Custody

Sometimes we need to transfer custody of a dynamic object from one place to another.

For example, a function might create an object and return a pointer to it. In this case, custody passes to the caller, since the creating function has given up custody when it returns.

Example:

```
Gate* makeGate(...) {
   return new Gate(...);
}
```

Custody of dynamic extensions

Similarly, with a shallow copy of an object with a dynamic extensions, there is an implicit transfer of custody of the dynamic extension from the old object to the new.

Problem: How does the old object give up custody? Possibilities:

1. Explicitly set the pointer to the dynamic extension in the old object to `nullptr`.
2. Destroy the old object.

The first is cumbersome and error-prone. The second causes a double-delete if the destructor does a `delete` of the dynamic extension.

Move versus copy

What we want in these cases is to **move** the object instead of copying it. The move first performs the shallow copy and then transfers custody to the copy.

Move semantics were introduced in C++ in order to solve this problem of transfer of custody of dynamic extensions.

Move Semantics

When to move?

With primitives, move and copy are the same. With large objects and objects with dynamic extensions, the programmer needs to be able to control whether to move or copy.

C++ has a kind of type called an **rvalue reference**.

An rvalue reference to a type `T` is written `T&&`.

Intuitively, an rvalue reference is a reference to a temporary. The actual semantics are more complicated.

Temporaries

Conceptually, a **pure** value is a disembodied piece of information floating in space.

In reality, values always exist somewhere—in variables or in temporary registers.

Languages such as Java distinguish between **primitive values** like characters and numbers that can live on the stack, and **object values** that live in permanent storage and can only be accessed via pointers.

A goal of C++ is to make primitive values and objects look as much alike as possible. In particular, both can live on the stack, in dynamic memory, or in temporaries.

Move semantics

An object can be moved instead of copied. The idea is that the data in the source object is removed from that object and placed in the target object. The source object is then said to be *empty*.

As we will see, what actually happens to the source object depends on the object's type.

For objects with dynamic extensions, the pointer to the extension is copied from source to target, and the source pointer is set to `nullptr`.

Any later attempt to delete `nullptr` is a no-op and causes no problems.

We say that **custody** has been transferred from source to target.

Motivation

A big motivation for move semantics comes from containers such as `vector`.

Containers need to be able to move objects around. Old-style containers can't work with dynamic extensions.

C++ containers support moving an object into or out of the container.

While in the container, the container has custody of the object.

Move is like a shallow copy, but it avoids the double-delete problem.

Implementation in C++

Here are the changes to C++ that enable move semantics.

1. The type system is extended to include **rvalue references**. These are denoted by double ampersand, e.g., `int&&`.
2. Results in temporaries are marked as having rvalue reference type.
3. A class has now six special member functions: constructor, destructor, copy constructor, copy assignment, move constructor, move assignment. These are special because they are defined automatically if the programmer does not redefine them.

Move and copy constructors and assignment operators

Copy and move *constructors* are distinguished by their prototypes.

`class T`:

- *Copy constructor:* `T( const T& other ) { ... }`
- *Move constructor:* `T( T&& other ) { ... }`

Similarly, copy and move *assignment operators* have different prototypes.

`class T`:

- *Copy assignment:* `T& operator=( const T& other ) { ... }`
- *Move assignment:* `T& operator=( T&& other ) { ... }`

Default constructors and assignment operators

Under some conditions, the system will automatically create default move and copy constructors and assignment operators.

The default copy constructors and copy assignment operators do a shallow copy. Object data members are copied using the copy constructor/assignment operator defined for the object's class.

The default move constructors and move assignment operators do a shallow copy. Object data members are moved using the move constructor/assignment operator defined for the object's class.

Default definitions can be specified or inhibited by use of the keywords `=default` or `=delete`.

Moving from a temporary object

A mutable temporary object always has rvalue reference type.

Thus, the following code *moves* the temporary string created by the on-the-fly constructor `string("cat")` into the vector `v`:

```
#include <string>
#include <vector>
vector<string> v;
v.push_back( string("cat") );
```

Forcing a move from a non-temporary object

The function `std::move()` in the `utility` library can be used to force a move from a non-temporary object.

The following code *moves* the string in `s` into the vector `v`. After the move, `s` contains the null string.

```
#include <iostream>
#include <string>
#include <utility>
#include <vector>
vector<string> v;
string s;
cin >> s;
v.push_back( move(s) );
```

The full story

I've covered the most common uses for rvalue references, but there are many subtle points about how defaults work and what happens in unusual cases.

Some good references for further information are:

- *Move semantics and rvalue references in C++11* by Alex Allain.
- *C++ Rvalue References Explained* by Thomas Becker.

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 12
October 8, 2018

Uses of Pointers

Feedback on Programming Style

Uses of Pointers

Array data member

A class `A` commonly relates to several instances of class `T`.

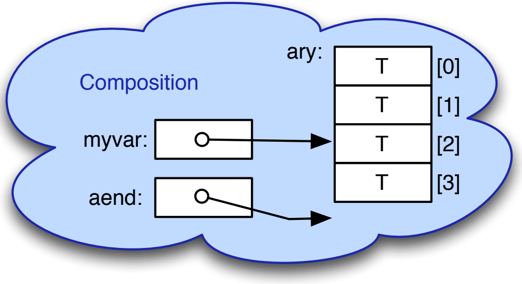
Some ways to represent this relationship.

1. **Composition:** `A` can **compose** an array of instances of `T`. This means that the `T`-instances are inside of each `A`-instance.
2. **Aggregation:** `A` can contain a pointer to a dynamically-allocated array of instances of `T`. `A` **composes** the pointer but **aggregates** the `T`-array to which it points.
3. **Fully dynamic aggregation:** `A` can contain a pointer to a dynamically-allocated array of *pointers* to instances of `T`. The individual `T`-instances can be scattered throughout memory.

Pictures of these three methods are given on the next slides.

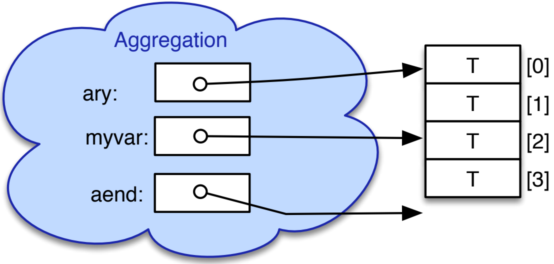
Composition

```
T   ary[4];
T* aend = ary+4;
T* myvar = &ary[2];
```



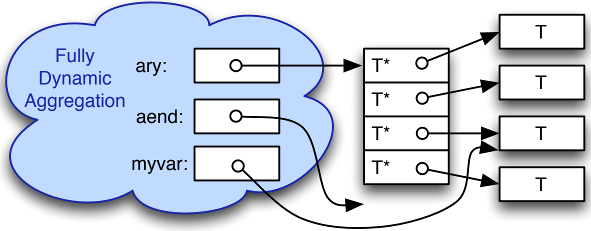
Aggregation

```
T* ary = new T[4];
T* aend = ary+4;
T* myvar = &ary[2];
```



Fully dynamic aggregation

```
T** ary = new T*[4];
T** aend = ary+4;
for( k=0; k<4; ++k ) {
    ary[k] = new T;
}
T* myvar = ary[2];
```



Pointer Arithmetic

Addition and subtraction of a pointer and an integer gives a new pointer.

```
int a[10];
int* p;
int* q;
p = &a[3];
q = &a[5];
// q-p == 2
// p+1 == &a[4];
// q-5 == &a[0];
// What is q-6?
```

Implementation

Pointers are represented internally by memory addresses.

The meaning of `p+k` is to add `k*sizeof *p` to the address stored in `p`.

Example: Suppose `p` points to a `double` stored at memory location 500, and suppose `sizeof(double) == 8`. Then `p+1` is a pointer to memory location 508.

508 is the memory location of the first byte following the 8 bytes reserved for the double at location 500.

If `p` points to an element of an *array* of `double`, then `p+1` points to the *next* element of that array.

Feedback on Programming Style

Coding Hints

In the next few slides, I will point out some miscellaneous programming issues that turned up on PS2. Proper C++ style is somewhat different from other languages (include C). Part of professional-level C++ proficiency is learning not just what works but what is simple and efficient.

Zero-tolerance for compiler warnings

Compiler warnings flag things that are not proper C++ usage but may work anyway in some environments. They generally indicate program errors or sloppy style.

You need to learn what the warnings mean and how to avoid them. Don't just ignore warnings because you think they are unimportant. "Unimportant" warnings will mask important ones that result from real bugs in your code.

Example: Comparing an `unsigned int` with an `int` gives such a warning.

Fix: Use appropriate integer types.

Declaration order in classes

There are two schools of thought on the order of declarations within classes:

1. Put the public functions first followed by the private.
   Rationale: The public functions represent the interface and are what clients of the class wnat to see.
2. Put the private data members and functions first followed by the public.
   Rationale: Generally names must be declared before they are used. It's natural to declare data members before functions that might use them, even if C++ provides some flexibility.

In this course, I require the second style: private first, public last.

Construct semantically consistent objects

Constructors should leave objects in a semantically meaningful state.

Avoid the paradigm common in other languages to create uninitialized objects and then initialize data members from member functions.

Use break

Instead of

```
bool exit = false;
while (!exit) {
  ...
  if (...) exit = true;
  else {
     ...
  }
}
```

use

```
for (;;) {
  ...
  if (...) break;
   ...
}
```

Use tolower()

Instead of

```
if (input=='Q' || input=='q') ...
```

use

```
#include <cctype>
...
input = tolower(input);
if (input=='q') ...
```

Use `switch`

Instead of

```
if (input=='a' || input=='b' || input=='c') { ... }
else if (input=='p') {
  ...
```

use

```
switch (input) {
case 'a':
case 'b':
case 'c': ...; break;
case 'p': ...; break;
}
```

Use stream input to read data

Instead of

```
int x;
string s;
s.getline(in);
// extract substring
// convert substring to number
 ...
```

use

```
int x;
in >> x;
```

Instead of

```
for (;;) {
   in >> x;
   if ( <error> ) {
      <handle error>
   }
   else {
      <do stuff>
      in >> y;
      if ( <error> ) {
         <handle error>
      }
      else {
        <do stuff>
      }
   }
}
```

Use `continue`

```
for (;;) {
   in >> x;
   if ( <error> ) {
      <handle error>
      continue;
   }
   <do stuff>
   in >> y;
   if ( <error> ) {
      <handle error>
      continue;
   }
   <do stuff>
}
```

Use `new` and `delete`, not `malloc` and `free`

C uses `malloc` and `free` to allocate and free dynamic storage.

C++ uses `new` and `delete`.

What are the differences?

1. `new` and `delete` are type safe; `malloc` and `free` are not.
2. `new` calls the constructor and `delete` calls the destructor. `malloc` and `free` are unaware of C++ classes and just handle uninitialized storage.
3. Array forms `new[]` and `delete[]` call default constructors and destructors of array elements.

Don't use `malloc` and `free` in C++ programs.

End-of-file handling

Don't use

```
while (!in.eof()) {
  in >> x;
  <do stuff with x>
}
```

to read and process a file of numbers. Even if `in.eof()` returns `false`, the next read might fail. Instead, use

```
for (;;) {
  in >> x;
  if (in.fail()) { <handle error/eof condition> }
  <do  stuff with x>
}
```

Include guards

Include guards are a method of using the C++ preprocessor to make sure that the declarations in a header file are not included more than once in a compilation. Here's how they work:

- A preprocessor symbol `GATE_HPP` is associated with a header file `gate.hpp`. Initially, `GATE_HPP` is undefined.
- Before `gate.hpp` is processed, `#ifndef GATE_HPP` is used to test if `GATE_HPP` is already defined.
- If it is, `gate.hpp` has already been processed and is skipped.
- If not, `#define GATE_HPP` defines `GATE_HPP` and the header file `gate.hpp` is processed.

Where do the include guards go?

They could be used to protect either the `#include "gate.hpp"` statement or the body of the header file `gate.hpp`.

Because there may be many `#include "gate.hpp"` statements in the program but there is only one `gate.hpp` file, they are normally placed inside the header file itself, e.g.,

```
// File gate.hpp
#ifndef GATE_HPP
#define GATE_HPP
   <body of header file>
#endif
```