Programming Principles and Practice Using C++ - Glossary of Terms

Brendan Duke 0770590

[1]Department of Physics and Astronomy, McMaster University,

*To whom correspondence should be addressed; E-mail: dukebw@mcmaster.ca.

September 17, 2014

Chapter 2

- // ≡ A comment.
- « ≡ The output operator.
- C++ ≡ a programming language based on C, which is capable of taking high-level code understandable by a programmer and translating it into machine code for the computer to process.
- Comment ≡ a line of code that the compiler will ignore, and which has the purpose of explaining the code to human readers.
- Compiler ≡ a piece of software that converts source code, the language of C++, into machine code readable by computers.
- Compile-time error ≡ an error that occurs during compiling and is detected by the compiler.
- **cout** ≡ instruction to print an input statement to a standard output stream. "Character **out**put stream".
- Executable ≡ a program that can be run by the computer directly.

- Function ≡ A named set of directions for the computer to execute in the order in which they're written. E.g. **main()** is a function
- Header ≡ a library of definitions of terms to be used by the compiler.
- IDE ≡ Interactive Development Environment; a text editor for code that has colour coding and debugging utilities, and a compiler and linker.
- **#include** ≡ an instruction to make available facilities from a specific file. E.g. the standard I/O facilities may be made available.
- Library ≡ a set of code accessed using declarations found in an **#include**d file.
- Linker ≡ links a set of separate object file parts (translation units) together into one executable file to be run by the computer. E.g. linking standard library files and the "Hello, World!" program.
- **main()** ≡ the function containing the main body of the program; this is where the computer starts executing instructions of a program.
- Object code ≡ Code in the form of machine code to be read by the computer.
- Output ≡ a type of information, e.g. an integer or a string, to be output by the program to the computer.
- Program ≡ a list of directions to be read and executed by a computer. E.g. the "Hello, World!" code written is a program.
- Source code ≡ the code written in a high-level language, e.g. written in C++, which can be converted into machine code and thus give the computer a set of instructions to execute. This is what you read and write.
- Statement ≡ a single line of code ending in a semi-colon or right curly bracket, e.g. cout « "Hello, World!\n";. These are the actions that a function is to perform. This is any action that is not an **#include** directive.

Chapter 3

- Assignment ≡ setting the value of a variable in the computer's memory, this is represented by "=".

- **cin** ≡ instruction to read in an input.
- Concatenation ≡ adding two things together, e.g. concatenating "cat" and "dog" to "cat-dog".
- Conversion ≡ changing from one type to another, e.g. from int to double.
- Declaration ≡ naming an object.
- Decrement ≡ stuff = stuff-1.
- Definition ≡ Introduces a new name into a program and sets aside memory for an object. A definition can and should assign an initial value. E.g. int count=0.
- Increment ≡ stuff = stuff+1.
- Initialization ≡ giving a variable its initial value, e.g. int count=0;.
- Name ≡ what a variable is called, e.g. "count" of int count=0;.
- Narrowing ≡ during conversion going from a type that has a large size to a type with a small size and hence losing information. E.g. char c = a, where a is an integer equal to 20000.
- Object ≡ A set of bits in memory with a type that stores information.
- Operation ≡ applying a set of rules to a given variable, e.g. multiplying 2 by itself is 2*2 and the operation is multiplication.
- Operator ≡ the character or string that invokes an operation, e.g. "*" of the multiplication 2*2.
- Type ≡ a size in bits of the information to be stored in memory and the associated rule for the meaning of those bits (the object), e.g. char, int, double, bool.
- Type safety ≡ keeping every object defined and used within its type, and not defining a type of smaller size in terms of one of larger size.
- Value ≡ the 1s and 0s stored in bits in memory and interpreted according to a type.
- Variable ≡ a named object.

Chapter 4

- Abstraction ≡ a description of how a program works in plain language, which is usually much shorter than the corresponding precise code.
- **begin()** ≡ member function that returns the beginning index and element of a vector.
- Computation ≡ alteration of an inputted value based on a set of rules, e.g. multiplying an inputted floating point value by two.
- Conditional statement ≡ a statement that is only executed if certain conditions are met, e.g. if(a==b) cout « a « " equals " « b « endl;.
- Divide and Conquer ≡ iteratively splitting a problem into smaller problems until they are tackle-able one at a time.
- **else** ≡ in the circumstance that none of the preceding conditions are met, do the following statements, e.g. else return x;.
- **end()** ≡ member function that refers to the last index and element of a vector.
- Expression ≡ performs a computation on a value.
- **for**-statement ≡ e.g. for(int i=0; i<100; ++i) will execute a statement 100 times as i goes from 0 to 99.
- Function ≡ a set of statements that takes in an input variable and returns an output, e.g. square(x) returns the square of x.
- lvalue ≡ object named by the left-hand operand of an assignment; the object being assigned a value, e.g. **length** of **length=12;**.
- Member function ≡ a function that acts on a vector and must be called using dot notation, e.g. size() of vector.size().
- **push_back()** ≡ a function that increases the size of the given vector by one and puts the thing in () at the end.
- Selection ≡ a choice based on comparison of a value against several constants, e.g. using **switch** with **case**s.
- **size()** ≡ returns the size of a vector; member function.

- **sort()** ≡ standard library function for sorting a vector.
- **switch**-statement ≡ divvies up cases based on whether a given character, integer or enumeration is equal to a number of different constant values.
- **vector** ≡ n by 1 array.

Chapter 5

- Argument error ≡ error due to incorrect input to the argument of a function.
- Assertion ≡ e.g. a==b. True/false statement. A statement that states (asserts) an invariant.
- **catch** ≡ function that defines what to do with a **throw**n error. Specifies what to do if the called code used **throw**.
- Compile-time error ≡ error, e.g. syntax error, that is caught by the compiler.
- Container ≡ e.g. a vector; set of bits in memory used to store data (elements).
- Debugging ≡ act of finding bugs/errors (undesirable parts) in a program.
- Error ≡ when a program does what it's not supposed to do/doesn't work.
- Exception ≡ definition of occurrence that causes a **throw**. It is what is **throw**n when a function finds an error that it cannot handle.
- Invariant ≡ statement that should always be true, e.g. speed < speed_of_light.
- Link-time error ≡ error that occurs during linking of modules of a program.
- Logic error ≡ when a program produces undesirable results that are not caught by the compiler or during linking. Logic errors must be caught by the programmer.
- Post-condition ≡ condition that must be met after an operation is performed, e.g. area >= 0.
- Pre-condition ≡ condition that must be met before an operation is performed, e.g.

```
if(length <= 0 || width <=0) error("...");
```

- Range error ≡ trying to write to a place in memory that is not reserved for a container, e.g. trying to write to v[v.size()] for a vector v.
- Requirement ≡ of a function; often called a pre-condition. It must be true for the function to perform its action correctly.
- Run-time error ≡ errors that aren't caught by the compiler or during linking.
- Syntax error ≡ error due to language in the program that is not part of C++, e.g. Cout « "Hello, world!" « endl;
- Testing ≡ the act of trying different inputs to a program to look for bugs.
- **throw** ≡ ending a **try** block upon a certain condition, and being caught by a **catch** block or ending the program.
- Type error ≡ trying to give the wrong type as input to a function, e.g. trying to input a string "Success!" to a function whose inputs are defined as integers.

Chapter 6

- Analysis ≡ deciding what the problem, inputs and outputs are. "What should be done". A set of requirements, or a specification.
- **class** ≡ user-defined type.
- Class member ≡ member function of a **class**. Declared inside the class definition; defined outside.
- Data member ≡ member function of data?.
- Design ≡ set of solutions to a problem, using the given inputs and outputs. Overall structure for the system: which parts the implementation should have and how those parts communicate. Decide which tools, e.g. libraries, help you structure the program.
- Divide by zero ≡ self-evident.
- Grammar ≡ a set of rules with which to group a set of inputs into sub-groups. Defines the syntax of our input.
- Implementation ≡ actual code made with the design. Write the code, debug it and test that it actually does what it is supposed to do.

- Interface ≡ way users interact with a **class**.
- Member function ≡ function that is defined within a class (type).
- Parser ≡ a program that reads a stream of tokens according to a grammar.
- **private** ≡ implementation of a **class**, which is not alterable by the user.
- Prototype ≡ first version of a project, may be simplified.
- Pseudo code ≡ steps of a code written in plain language.
- **public** ≡ user-interface of a **class**.
- Syntax analyzer ≡ see parser. A program that reads a stream of tokens according to a grammar.
- Token ≡ user-defined type with a set of bits for **char** and a set of bits for **double**. A sequence of characters that represents something we consider a unit, such as a number or operator.
- Use case ≡ set of examples that a program should be able to solve.

Chapter 7

- Code layout ≡ the order of a code and its splitting into functions. Also, the way the code is presented in the program: how the lines are spaced, and how statements and parts of statements are separated on each line. E.g. the code layout of,

```
switch(ch){
case
   'q':case';':case'%':case'(':case')':case'+':case'-':case'*':case'/':
      return Token(ch); // let each character represent itself
```

could be better represented using one line per case and adding a couple comments. E.g.,

```
Token Token_stream:: get()
      // read characters from cin and compose a Token
{
      if (full) { // check if we already have a Token ready
            full=false;
            return buffer;
```

```
        }
        char ch;
        cin >> ch;  // note that Âż skips whitespace (space,
           newline, tab, etc.)
        switch (ch) {
        case quit:
        case print:
        case '(':
        case ')':
        case '+':
        case '-':
        case '*':
        case '/':
        case '%':
              return Token(ch); // lei each character represent
                 itself
        case 'âĂć': // a floating-point-literal can start with a
           dot
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
           //numeric literal
        {     cin.putback(ch); //put digit back into the input
           stream
              double val;
              cin >> val; // read a floating-point number
              return Token(number,val);
        }
        default:
              error("Bad token");
        }
}
```

- Commenting ≡ Non-code parts for the programmer to see that express intention and explain parts of the code that aren't better explained in code.
- Error handling ≡ detection and actions taken upon detection of errors in a program. Trying to feed a program input in the hope of getting it to misbehave. "I'll break it! I'm smarter than any program - even my own!" attitude.
- Feature creep ≡ the tendency to want to add many features. A tendency to add extra functionality to a program "just in case".

- Maintenance ≡ Keeping a code updated and working after it has been tested to a stable version.
- Recovery ≡ after a program detects an error, its switching back to perform its main function.
- Revision history ≡ a log of all the changes made to a program over time. Contains indications of what corrections and improvements were made.
- Scaffolding ≡ structure of a program: how it is split into functions and which functions call which other functions. E.g. start the program, end the program, and handle "fatal" errors. # includes etc.
- Symbolic constant ≡ E.g. defining,

```
const char number = '8';
```

  avoids the use of "magic constants". A symbolic name for the constant used to represent something (e.g. a number).
- Testing ≡ trying good and bad inputs on a program in order to try to make it crash, and hence find bugs. Breaking programs. E.g. Bjarne Stroupstrup fed e-mail reporting compiling errors straight to the compiler (when testing compilers).

Chapter 8

- Activation record ≡ A data structure that is a detailed layout of each function. A function's parameters, local variables and implementation are stored in memory when the function is called.
- Argument ≡ a value passed to a function or a template, in which it is accessed through a parameter.
- Argument passing ≡ process by which an argument is parameterized for use by a function. E.g. pass-by-value, pass-by-reference and pass-by-**const**-reference.
- Call stack ≡ A data structure that grows and shrinks at one end according to the rule: first in, first out. A sequence of instructions in the memory that are destroyed in a last-in-first-out order.

- Class scope ≡ The area of text within a class. The scope for which a given class is defined and usable.
- **const** ≡ a command associated with an object by the compiler stating that it cannot be re-written to. (Constant: a value that cannot be changed (in a given scope); not mutable.)
- **constexpr** ≡ **constexpr** functions are functions that are evaluated by the compiler. A **constexpr** function can be evaluated by the compiler if it is given constant expressions as arguments. A **constexpr** function behaves just like an ordinary function until you use it where a constant is needed; then, it is calculated at compile time provided its arguments are constant expressions (and gives an error if they are not). A **constexpr** function must be so simple that in C++11 it must have a body that consists of a single return statement; in C++14, we can also write simple loops. A **constexpr** function may not change the values of variables outside its own body, except those it is assigned to or uses to initialize.
- Declaration ≡ A statement that introduces a name into a scope. A declaration could be specifying a type for what it is named (e.g., a variable or a function.) Or, a declaration could optionally be specifying an initializer (e.g., an initializer value or a function body). Naming of a variable, function or class.
- Definition ≡ A declaration that also fully specifies the entity declared. Naming of a variable, function or class and setting aside memory for it. A definition gives the implementation of a function, assigns a value to a variable and gives the implementation of a class.
- **extern** ≡ Extern plus no initializer means "not definition. It is rarely useful. It is recommended that you don't use it, but you'll see it in other people's code, especially code that uses too many global variables.
- Forward declaration ≡ a declaration that is not a definition, declared before its definition in order that it can be used recursively.
- Function ≡ A named unit of code that can be invoked (called) from different parts of a program; a logical unit of computation. A set of variables and statements.
- Function definition ≡ the body of a function - contains statements and variables used within the function.
- Global scope ≡ The scope that's not nested within any other, or the area of text outside any other scope. The scope that all other scopes are nested within.

- Header file ≡ A collection of declarations, typically defined in a file (hence a header is also called a *header file*). A file containing declarations and definitions that can be **#include**d in other files.
- Initializer ≡ An initializer value or function body. What a variable is set to when it is defined. An objects first (initial) value.
- Local scope ≡ Between {...} braces of a block or in a function argument list. The scope within a block - set by brackets {}.
- **namespace** ≡ The language mechanism for organizing classes, functions, data and types into an identifiable and named part of a program without defining a type. A place for defining and declaring functions, classes and variables that does not have a type.
- Namespace scope ≡ A named scope nested in the global scope or in another namespace. The code within a **namespace** for which its declared functions, variables and classes apply.
- Nested block ≡ blocks within functions and other blocks, e.g.

```
{
        // ...
        {
                // ...
        }
}
```

- Parameter ≡ A declaration of an explicit input to a function or a template. When called, a function can access the arguments passed through the names of its parameters. Formal arguments. Argument passed to a function.
- Pass-by-**const**-reference ≡ an argument to a function that is a reference to an object, i.e. points to the object itself, and that object is not allowed by the compiler to be altered by the function referencing it.
- Pass-by-reference ≡ passing an argument to a function that is a reference to an object, i.e. points to an object's location in memory.
- Pass-by-value ≡ A copy of an object is made for use within the scope of the function being passed to.
- Recursion ≡ when a function, directly or indirectly, calls itself.

- **return** ≡ When a function returns, its activation record is no longer used. Signifies where a function deletes all its implementation, local variables etc. that it built up on the stack.
- Return value ≡ the value a function puts on the stack when it **return**s.
- Scope ≡ A scope is a region of program text. A name is declared in a scope and is valid (is "in scope") from the point of its declaration until the end of the scope in which it was declared. E.g.

```
void f()
{
      g();  // error: g() isn't (yet) in scope
}

void g()
{
      f();  // OK: f() is in scope
}

void h()
{
      int x=y;     // error: y isn't (yet) in scope
      int y=x;     // OK: x is in scope
      g();         // OK: g() is in scope
}
```

  Functions within which variables, classes, function etc. definitions apply.

- Statement scope ≡ the scope within a statement, e.g.

```
for(int i=0; i<v.size()-1; ++i){
      v[i] += v[i+1];
}
```

  "i" is defined within the brackets of this **for** statement only.

- Technicalities ≡ specific details of a language that don't give much over-arching programming understanding.
- Undeclared identifier ≡ Compilers will give an error for "undeclared identifiers", since they are not declared anywhere in a given program fragment. A name used that hasn't been declared.

- **using** declaration ≡ E.g. the constructs,

```
using std::string; // string means std::string
using std::cout;  // cout means std::cout
//...
```

  the programming equivalent to using plain "Greg" to refer to Greg Hansen, when there are no other Gregs in the room.

- **using** directive ≡ E.g.

```
using namespace std; // make names from std directly accessible
```

  For the names from a namespace: "If you don't find a declaration for a name in this scope, look in **std**."

Chapter 9

- built-in types ≡ Types that the compiler already knows how to set aside memory (an object) for, and the definitions of the allowed operations on said default types (also whether they're allowed, e.g. + and *) without being told by declarations supplied by a programmer in source code.
- **Class** ≡ A user-defined type. Members are private by default. Classes may contain data members, function members and member types.
- **const** ≡ **const** declares that an object (?) can't be altered. Used to define symbolic constants.
- constructor ≡ code that specifies initialization of an object of a type. A member function with the same name as its class. It is an error - caught by the compiler - to forget to initialize an object of a class that has a constructor that requires an argument.
- destructor ≡ an operation that is implicitly invoked (called) when an object is destroyed (e.g., at the end of a scope). Often, it releases resources
- **enum** ≡ A used-defined type that specifies its values (its enumerators) as symbolic constants. The "body" of an enumeration is simply a list of enumerators. For an **enum class**, the enumerators are in the scope of the enumeration. If you leave it to the compiler to specify values, it'll give each enumerator the value of the previous enumerator plus one (starting with zero if unspecified).

- enumeration ≡ See **enum**.
- enumerator ≡ See **enum**.
- helper function ≡ A function, possibly in a namespace of related functions, that provides some functionality to a Class and is not a member function of that class. Helper functions often take arguments of the classes that they are helpers of. "Helper function" is a design concept.
- implementation ≡ Hidden from the rest of the program outside its Class, a Class' implementation "implements" that class. (**private:**). The implementer's view of the class.
- in-class initializer ≡ An initializer for a class member that is specified as part of the member declaration. E.g.,

```
class Date {
public:
      //...
      Date();            // default constructor
      Date(year, Month), day);
      Date(int y); // January 1 of year y
      // ...
private:
      int y {2001};
      Month m {Month::jan};
      int d {1};
};
```

- inlining ≡ if a function is **inline**, the compiler will try to generate code for the function at each point of call rather than using function-call instructions to use common code. This can be a significant performance advantage for functions, such as **month**(), which hardly do anything and are used a lot. Writing the definition of a member function within the class definition makes the function **inline**.
- interface ≡ The **public:** part of a **class**, which is used by and visible to the user. The user accesses the implementation of a class only indirectly through the interface.
- invariant ≡ A rule for what constitutes a valid value, e.g. a **Date** must lie in the past, present or future. Something that must be always true at a given point (or points) of a program; typically used to describe the state (set of values) of an object or the state of a loop before entry into the repeated statement.

- representation ≡ A type "knows" how to represent the data needed in an object.
- **struct** ≡ A class for which the default members are public. **struct**s are primarily used for data structures where the members can take any value; that is we can't define any meaningful invariant.
- structure ≡ A data structure, e.g. data members in a **class**.
- user-defined type ≡ Inherently obvious.
- valid state ≡ See invariant. E.g. the allowed months, years and days of a **Date**.

Chapter 10

- **bad()** ≡ stream state that indicates that the stream has suffered an error that is unlikely to be recoverable from. Something unexpected and serious happened (e.g., a disk read error).
- buffer ≡ holds values as they are inputted to an input stream, or outputted from an output stream. A data structure that the **ostream** uses internally to store the data you give it while communicating with the operating system.
- **clear()** ≡ member function of a stream state that resets its state to **good()**. **clear()** explicitly takes a stream out of the **fail()** state.
- **close()** ≡ member function of I/O streams that closes a file.
- Device driver ≡ provides a software interface to hardware devices, enabling an operating system to use hardware functions without knowing the precise details of the hardware being used.
- **eof()** ≡ stream state indicating that the end of file has been reached.
- **fail()** ≡ stream state indicating that an error has occurred, which is likely recoverable from. Something unexpected happened (e.g., we looked for a digit and found 'x').
- File ≡ a numbered sequence of bytes. A container of permanent information in a computer.
- **good()** ≡ stream state indicating that the stream is good; the operations succeeded.
- **ifstream** ≡ input-file stream. An **ifstream** is an **istream** for reading from a file.

- Input device ≡ a device that produces input (e.g. a keyboard).
- Input operator ≡ "get-from":

  >>

  ---

  can be defined for user-defined types.
- **iostream** ≡ A stream used for both input and output.
- **istream** ≡ stream used for input. An **istream** turns character sequences into values of various types, and gets those characters from somewhere (such as a console, a file, the main memory or another computer).
- **ofstream** ≡ an **ostream** that is used to write to files.
- **open()** ≡ member function of a stream used for opening a file.
- **ostream** ≡ stream used for output. An **ostream** turns values of various types into character sequences, and and sends those character sequences "somewhere" (such as to a console, a file, the main memory or another computer).
- **output device** ≡ a device used for output, e.g. a monitor or a text window.
- **output operator** ≡ "put-to":

  <<

- Stream state ≡ consists of four possible cases: **good()**, **bad()**, **eof()** and **fail()**, as defined above.
- Structured file ≡ A file with some sort of format, such as the temperature format {year 1992 {month jan (1 0 61.5)}}.
- Terminator ≡ Character that indicates that reading of a file should stop.
- **unget()** ≡ puts the last character to be taken out of the input stream back into the input stream. A shorter version of **putback()**, which relies on the stream remembering which character it last produced so that you don't have to mention it.

Chapter 11

- Binary ≡ sets of 1s and 0s used to store information on a computer. A set of bit values are in binary.
- Character classification ≡ e.g. classifying a character as alphanumeric, a digit, punctuation, etc.
- Decimal ≡ base-10 number representation.
- **defaultfloat** ≡ The 6-digit precision default number-representation setting, which chooses to use **scientific** or **fixed** based on which is more accurate (within the precision of **defaultfloat**).
- File positioning ≡ Where the cursor is in a file, i.e. which byte number will be written to or read from next. Every file open for reading has a "read/get position", and every file open for writing has a "write/put position". **seekg()** and **seekp()** increment the reading and writing positions, respectively.
- **fixed** ≡ manipulator that specifies that the output is to have 6 digits of precision after the decimal.
- Hexadecimal ≡ base-16.
- Irregularity ≡ not treating all input sources as equivalent. Regularity would imply imposing a single standard on the way to treat objects entering and exiting the system.
- Line-oriented input ≡ processing of input that is done by taking one line at a time using **getline**, instead of breaking input up with whitespace.
- Manipulator ≡ a command that gives get-from or put-to operators some kind of setting, e.g. **std::hex** or **std::left**. Terms that are used to change the behaviour of a stream.
- Nonstandard separator ≡ using something besides whitespace as a separator, e.g. punctuation ".;:" etc.
- **noshowbase** ≡ manipulator that tells an output stream not to show the base of octal and hexadecimal numbers, e.g. 0x (or 0X) or 0 are emitted in front of the number.
- Octal ≡ base-8 notation.
- Output formatting ≡ e.g. precision of outputted floating point numbers.

- Regularity ≡ see irregularity.
- **scientific** ≡ manipulator that makes floating-point numbers output with a mantissa in [0:10), and with six digits of precision after the mantissa.
- **setprecision()** ≡ sets the precision of a floating-point number.
- **showbase** ≡ shows the base of hex and octal numbers in output, e.g. 02322 or 0x4d2.

Chapter 12

- Color ≡ e.g. red. A class in the graphics library used.
- Coordinates ≡ a set of numbers specifying points in space (of some number of dimensions).
- Display ≡ a display model is the output part of GUI.
- Fill color ≡ Fill a Shape with a Color.
- FLTK ≡ Fast Light ToolKit, pronounced "fulltick".
- Graphics ≡ visible classes (?) such as Shapes, Line, Lines, Polygons etc.
- GUI ≡ Graphical User Interface. The graphical interface between a user and a program.
- GUI library ≡ a program, which we can think of as the "display engine" - or "the small gnome writing on the back of the screen", which takes objects attached to a window and draws them on the screen.
- HTML ≡ A layout (typesetting, "markup") language. HyperText Markup Language: the standard markup language used to create web pages.
- Image ≡ e.g. a jpeg or a gif file.
- JPEG ≡ a commonly used method of lossy compression for digital images. The degree of compression can be adjusted.
- Line style ≡ e.g. dash, dash-dot-dot, etc.

- Software layer ≡ e.g. a GUI. In object-oriented design, a layer is a set of classes that have the same set of link-time module dependencies to other modules. In the GUI program of Chapter 16, examples of layers were *Our program*, *Our graphics/GUI interface library*, *FLTK*, *The operating system graphics/GUI facilities*, and the *Device driver layer*.
- Window ≡ an object that other objects can be attached to and thereby displayed.
- XML ≡ Extensible Markup Language: a layout (typesetting, "markup") language. Defines a set of rules for encoding documents that is both human-readable and machine-readable.

Chapter 13

- Closed shape ≡ A shape whose last point is connected to its first point, and all points in between are connected.
- Color ≡ e.g. red. The RGB color model is a model in which red, green and blue light are added together in various ways to produce a broad array of colors.
- Ellipse ≡ $x^2/a^2 + y^2/b^2 = 1, \quad a, b \in \mathbf{P}$. a and b are the semi-major and semi-minor axes.
- Fill ≡ to fill with color.
- Font ≡ e.g. Helvetica.
- Font size ≡ e.g. size 14. Height of a font in pixels.
- GIF ≡ "Graphics Interchange Format": a bitmap image format that supports up to 8 bits per pixel for each image, allowing a single image to reference its own palette of up to 256 different colors chosen from the 24-bit RGB color space.
- Image ≡ A file that is a set of connected pixels, e.g. a JPG or a GIF, a picture of Hurricane Rita etc.
- Image encoding ≡ the method used to store an image, and also the file-extension given to that image. The object of image encoding is to reduce redundancy in image data in order to be able to store or transmit image data in an efficient form.
- Invisible ≡ sets the pixels in a graphic to transparent.

- Line $\equiv$ defined by two points, or $y = mx + b$.
- Open shape $\equiv$ a shape for which any two consecutive points are not connected.
- Point $\equiv$ defined by x and y pixels; a single pixel on the screen.
- Polygon $\equiv$ a **Closed_polyline** for which no two lines intersect.
- Polyline $\equiv$ A set of lines for which each subsequent line has a point that is shared with the previous line.
- Unnamed object $\equiv$ an object for which space is set aside in the heap ("free store"), and for which there is no name to reference it.
- **Vector_ref** $\equiv$ a container for collections of objects of different types. A **vector** type that can hold both named and unnamed objects. It is used much like a standard library **vector**.
- Visible $\equiv$ pixels that are not fully transparent. 256 values (8-bit).

Chapter 14

- Abstract class $\equiv$ a class for which objects of that class are not intended or able to be created. An abstract class can be used only as a base class; often used to define an interface to derived classes. A class is made abstract by having a pure virtual function or a protected constructor.
- Access control $\equiv$ achieved using **public:**, **private:**, and **protected:**, access control determines which functions can access member functions and data of a class.
- Base class $\equiv$ a class from which other classes are derived, whose member functions and data are often passed on to the derived classes. A class used as the base of a class hierarchy. Typically a base class has one or more virtual functions.
- Derived class $\equiv$ a class that gets the member functions of its base class, declared using,

```
struct Derived : Base {
      // ...
};
```

- Dispatch ≡ run-time polymorphism, dynamic dispatch or run-time dispatch. The ability to define a function in a base class and have a function of the same name and type in a derived class called when a user calls the base class function. E.g. when **Window** calls **draw_lines** for a **Shape** that is a **Circle**, it is the **Circle**'s **draw_lines** that is executed rather than **Shape**'s own **draw_lines**.
- Encapsulation ≡ hiding data members using access control such as **private:** and **protected:**.
- Inheritance ≡ Derived classes getting their member functions and **public:** data members from their base classes. A.k.a. derivation: a way to build one class from another so that the new class can be used in place of the original. E.g., **Circle** is a kind of **Shape**.
- Mutability ≡ changeability; the opposite of immutable, constant and variable. We try to ensure that the state of an object is only mutable by its own class.
- Object layout ≡ how the data members and member function information are laid out in an object. Members of a class define the layout of objects: data members are stored one after another in memory. When inheritance is used, data members of a derived class are added after those of a base.
- Object-oriented override ≡ overriding **virtual** functions of base classes with same-named functions of derived classes. Declared with **override**.
- Polymorphism ≡ See dispatch.
- **private** ≡ data or functions only available to member functions of a class.
- **protected** ≡ data or functions available to member functions of a class and of its derived classes.
- **public** ≡ data or functions that can be used by any function.
- Pure virtual function ≡ a function of a base class that makes that class abstract, and that must be overridden by a same-named function in a derived class for that class to be made concrete.
- Subclass ≡ derived class.
- Superclass ≡ base class.

- Virtual function ≡ a function in a base class that is to be overridden in its derived classes. The ability to define a function in a base class and have a function of the same name and type in a derived class when a user calls the base class function. See dispatch.
- Virtual function call ≡ a call of a virtual function, which is handled using a piece of data in a derived class that tells which function is really invoked by that virtual function call (a function of some class in the class hierarchy is invoked). This is done using an address to a virtual function table or **vtbl**, which is a table of functions. The **vtbl**'s address is often called the **vptr** (for "virtual pointer").
- Virtual function table ≡ see virtual function call.

Chapter 15

- Approximation ≡ estimation of an exact value, e.g. by approximating e^x with a truncated Taylor series.
- Default argument ≡ of a function, a value that an argument will take if that argument is not provided, e.g. **xscale** and **yscale** in the constructor for **Function**,

```
struct Function : Shape {
// the function parameters are not stored
      Function(Fct f, double r1, double r2, Point orig,
            int count = 100, double xscale = 25, double
               yscale = 25);
};
```

  Default arguments are initializers given to contructor arguments in their declaration.
- Function ≡ a rule for transforming a set (the domain) into another set (the range). Each **Function** specifies how its first argument (a function of one **double** argument returning a **double**) is to be drawn in a window.
- Lambda ≡ a way of passing a function as an argument to another function, wthout declaring the function to be passed earlier in the code. C++ offers a notation for defining something that acts as a function in the argument position where it is needed. E.g.,

```
Function s5{[](double x){ return cos(x) + slope(x); }, r_min,
      r_max, orig, 400, 30, 30};
```

- Scaling ≡ multiplying all values in a set by a constant to achieve some purpose, such as making a set of data fit on a pair of axes of a certain size.
- Screen layout ≡ the way all different objects are laid out on the screen. E.g., how a pair of axes is offset from the top, bottom and sides of the screen etc.

Chapter 16

- Button ≡ a GUI object on the screen to be clicked by the mouse, that may be linked to some action. E.g. a class in the GUI interface to FLTK that is a subclass of a **Widget**. Buttons have a "Callback" parameter.
- Callback ≡ a low-level pair of addresses that is a way from getting from an action detected by device drivers from a device (e.g. a mouse-click), to a C++ function. A callback function is a function that we want the GUI system to call when it detects a click on our button.
- Console I/O ≡ output taken as strings (?) from a console, e.g. a Unix terminal or the MS/DOS terminal. This is strong for technical/professional work where the input is simple and textual. The C++ standard library **iostreams** provide suitable means to display output text on the screen or store it in files.
- Control ≡ a more descriptive, but less evocative, name for a widget.
- Control inversion ≡ when instead of the program following a flow controlled by the program's layout, callbacks from devices such as the mouse or keyboard control the program flow. When control of the order of execution of a program is moved to widgets, whichever widget the user activates runs. E.g., click on a button and its callback runs. When that callback returns, the program settles back, waiting for the user to do something else.
- Dialog box ≡ a box with input or output text in it. A small window that communicates information to the user and prompts them for response.
- GUI ≡ Graphical User Interface. A GUI allows a user to interact with a program by pressing buttons, selecting from menus, entering data in various ways, and displaying textual and graphical entities on a screen.
- Menu ≡ a subclass of a **Widget** that has multiple **Buttons** attached to it, all of which may have callbacks to different functions in the program.

- User interface ≡ an interface between the user and a computer program, e.g. windows and buttons that can be clicked by the mouse, and dialog boxes that can have text input or output to/from them. The user interface on a small gadget may be limited to input from a couple of push-buttons and output to an LED. Other computers are connected to the outside world only by a wire.

- Visible/hidden ≡ Drawn on the screen or not. **show()** and **hide()** member functions control visibility for **Widgets**.

- Waiting for input ≡ when the program is in a **while();** state awaiting callbacks from different widgets in order to decide which functions to run. In the GUI example of Chapter 16, **while(wait());** tells the system to look out for widgets and invoke the appropriate callbacks.

- Wait loop ≡ see explanation of **while(wait());** above.

- Widget ≡ a mother class for different objects, such as **Buttons**, that can invoke callbacks to functions within the program from GUI system-level device inputs. A **Widget** is a GUI entity for which a callback is triggered. For each action on a **Widget**, two functions have to be defined: one to map from the system's notion of a callback and one to do our desired action.

- Address ≡ the address of a memory location. Number that specifies a memory location. Can think of it as a kind of integer value.

- Allocation ≡ telling the operating system to reserve some memory (from free store) at a given address to be used by your program.

- Cast ≡ e.g. **const_cast** casts away const-ness or **reinterpret_cast** casts away the type of a value or pointer, or **static_cast** explicitly converts between pointer types. Explicit type conversion.

- Container ≡ e.g. a **std::vector**: an object that holds other objects.

- Contents of: * ≡ dereference operator: says to return the value of the object pointed to by a pointer.

- Deallocation ≡ telling the operating system to "free up" previously allocated memory.

- **delete** ≡ De-allocate a single object pointed to by a pointer.

- **delete[]** ≡ De-allocate an array of objects pointed to by a pointer.

- Dereference ≡ See contents of: *****.
- Destructor ≡ Function that is automatically called when an object goes out of scope; does the opposite of a constructcor. A destructor (should) make sure that any memory allocated by the object is de-allocated when it goes out of scope.
- Free store ≡ memory that is not used by the code, for static storage, or for the stack. Also known as the heap.
- Link ≡ an object that has at least one pointer to another object. E.g. with a double-link we can get to its successor with a **succ** pointer and to its predecessor (in the list) with a **prev** pointer.
- Member access: **->** ≡ like the dot **.** notation for object names: used to access members of the object pointed to by a pointer.
- Member destructor ≡ destructor of the member of an object. Destructors are called in the following order for a derived class: first the destructor of that object's class is called, then its member destructors are called in the reverse order in which they were declared in its class structure. Then the destructors of the derived class's base classes are called in reverse order. I.e. if **C1** derives from **B1**, and **B1** derives from **A1**, **C1**'s constructor is called, then all of **C1**'s member destructors are called, then **B1**'s destructor is called, all **B1**'s member destructors are called, then **C1**'s destructor is called, etc.
- Memory ≡ space for storing bits in the hardware. Made up of code, static storage, free store and the stack.
- Memory leak ≡ when, e.g., memory is allocated at an address pointed to by a pointer and then that pointer is made to point to a different address without first deallocating that memory.
- **new** ≡ tells the OS to allocate memory on the heap..
- Null pointer ≡ a pointer that doesn't point to any address, aka **NULL** or **0** or **nullptr**.
- **nullptr** ≡ see Null pointer.
- Pointer ≡ a type that contains a memory address, which can be no memory address (**nullptr**). Arithmetic can be done on pointers, and a pointer can be changed to point somewhere else (unless it has been declared **const**).

- Range ≡ e.g. of an array. The memory allocated to a container; any memory beyond that allocation is out of range.
- Resource leak ≡ see memory leak.
- Subscripting ≡ e.g. **a[n]** of an array **a** will refer to the **n**th object of type **a[0]** starting from the address pointed to by a[0].
- Subscript **[]** ≡ see subscripting.
- **this** ≡ points to the current object, e.g. in a member function,

```
Link* Link::insert(Link* n)
{
      if (n==nullptr) return this;
      // ...
}
```

  this refers to the **Link*** for the object for which **Link::insert** is being called (i.e. the **Link** object for which **insert()** is a member).

- Type conversion ≡ e.g. from **int** to **int*** (which requires a **static_cast**.
- **virtual** destructor ≡ a destructor of a base class that can be "replaced" by the destructor of its derived class. E.g. if a **Shape** that is a **Text**, and it goes out of scope, then when **Shape()** is called, since **Shape()** is virtual, the call invokes the destructor of **Shape**'s derived class, in this case **Text()**.
- **void*** ≡ a void pointer; a type-less pointer. Means "pointer to some memory that the compiler doesn't know the type of".