

Section Notes 6

CS51—Spring, 2008

Week of March 16, 2008

1 C++ and Scheme

C++ and Scheme solve problems in different ways—you can write Scheme-like code in C++, but you probably don't want to. We'll start using things like variables, classes, and explicit looping. On the other hand, you'll still need to think in terms of abstractions and must practice good style. (In fact, as the complexity of your code will grow, you should practice better style.)

1.1 C++ File Organization

In Scheme, you were used to having a single file containing all of your code. In C++, this is generally not the case. Instead, you will break down your code into header files (files ending in `.h`) and source code files (ending in `.cc`). You will also see people use the extensions `.hpp` and `.cpp` for these kinds of files. Header files should contain things like class declarations, constants, and function declarations for functions that are not part of a class.

You may find it cleanest to have a pair of `.h` and `.cc` files for each class. In assignment 5, we have provided a clean file organization for you.

1.2 Incremental Coding and Testing

In Scheme, you had the luxury of testing individual lines of code in an interpreter. In C++, you have to do a bit more work to build incrementally. You will also likely spend more time looking things up in man pages or online references. Nevertheless, you can test C++ constructs by creating small test programs that perform a few operations and display the results. Another technique is to write each class separately and test it before writing another class. In this way, you can isolate bugs to only the latest class being developed if you are reasonably confident in your testing procedures. This approach is similar to building and testing each function separately in Scheme.

2 Classes

Classes capture the idea of having a single entity both store data and store the means of operating on that data. Classes are a tool for providing a language-enforced abstraction barrier by keeping certain data and functions from being accessed outside of the class.

Class definitions provide a blueprint for class instances called objects. Each object possesses its own copies of the variables defined in the blueprint (“class definition”).

2.1 Syntax

Classes are syntactically similar to structs. You can define fields for your class the same way you would with a struct:

```

class Student {
    public:
        int gradeLevel;
        double gpa;
};

```

Notice that you must still terminate a class definition with a semicolon. Unlike structs, however, the data within a class is not by default accessible to the outside world; you must explicitly tell the compiler to allow access to fields by using the `public` keyword.

Classes, unlike C structs, also have member functions (“methods”) that can be called and that have access to the data fields of the class. Fields marked `private` are only visible to these methods.

Methods often should be defined in a separate `.cc` file. This necessitates telling the compiler which class a particular function declaration belongs to. The scope operator, `::`, will accomplish this for us. For instance, if we decided that the student’s `gpa` should be hidden from outside the `Student` class, we might want to add a method that would allow access to some data derived from `gpa`. For instance, we might include a `rank()` method that gives us the student’s ranking percentile.

```

class Student {
    private:
        int gradeLevel;
        double gpa;

    public:
        double rank();
};

```

Now, in a separate `.cc` file, we define the `rank()` method using the following syntax:

```

double Student::rank()
{
    // compute the rank in some way...
    return gpa / 10;
}

```

2.1.1 Members and Methods

There is one particular sort of method that we expect you to become intimately familiar with in writing C++ code. With C structs, all of the members were public and could be accessed by the `.` or `->` operators. In C++, one should make members private and provide access to data through getter and setter methods. These methods formalize the separation between interface and implementation and also allow the programmer to restrict access to data members as appropriate - maybe some data should be read and not modified, for example. For our `Student` class, we would add methods such as:

```

double Student::getGpa()
{
    return gpa;
}
void Student::setGpa(double newGpa)
{
    gpa = newGpa;
}

```

These accessor methods can be used within the implementation of our class (e.g., consider how we would now rewrite our `rank` method), or may be called by other code that uses objects of our class.

2.2 Object Life Cycle

2.2.1 Constructors

Just as you shouldn't use uninitialized variables, you wouldn't want new instances of your class to contain uninitialized values. In C++, whenever an object is declared, the act of declaration also invokes a function known as the constructor. Inside the constructor, all of the necessary class fields can be initialized.

We declare a constructor by using the name of the class as the name of the constructor. Note that constructors do not have a return value. (Where would it go, anyway?)

For instance, we might want to add a constructor to our `Student` class to set `gradeLevel` and `gpa`:

```
class Student {
    private:
        int gradeLevel;
        double gpa;

    public:
        double rank();
        Student(double, int);
};
```

and in the `.cc` file:

```
Student::Student(double initGpa, int initGradeLevel)
{
    gpa = initGpa;
    gradeLevel = initGradeLevel;
}
```

2.2.2 Destructors

Good programmers not only initialize their variables before use, but clean up after themselves. C++ allows you to provide a function, known as a destructor, that runs whenever an object is destroyed. (For instance, if a local variable goes out of scope or a pointer to an object is `delete`'d.)

Destructors are declared with the name of the class preceded by a tilde:

```
class Student {
    private:
        int gradeLevel;
        double gpa;

    public:
        double rank();
        Student(double, int);
        ~Student();
};
```

Destructors do not take any arguments and have no return value (where would it go?). Generally, destructors are used to free memory, close files and gracefully release any other resources that had been in use.

2.3 Class-wide Variables (“Static Class Variables”)

Most of the fields in your classes will belong to a particular object, but sometimes it is useful to have a variable that is shared by every object of a class (in fact, a variable that doesn't require the existence of any object in a class). Such variables are called static variables. Any object will be able to refer to or set the

static variable and changes will be visible to all other objects in the class. (Depending on how access to the static variable is defined, it may also be accessible outside of the class completely independent of whether there are any objects of the class.)

The syntax for a static variable is simply to prefix the variable with the keyword `static` (don't be confused—you may remember from CS 50 that static can also be used in other contexts).

```
class Student {
    private:
        int gradeLevel;
        double gpa;
        int studentId;
        static int nextStudentId;

};

int Student::nextStudentId = 0;
```

Notice that in the above class, we've added a `studentId` field (which will be different for each object) and a static field `nextStudentId`, which belongs to the class as a whole, and which we can use to set the `studentId` for each object we create.

Notice that to initialize the static field requires giving the type of the field, the name of the class and using the scope operator to identify which static field is being accessed.

As a rule of thumb, you need to use the scope operator, `::`, whenever you are working with a class, and the dot (`.`) or arrow (`->`) operators when working with a particular instance of a class.

2.4 Declaring Objects

Declaring an object is fairly simple. If you have a no-argument constructor, the syntax is almost the same as declaring a struct:

```
Student aStudent;
```

Note to C programmers: declaring structs in C++ is similar to declaring class instances—you can drop the keyword `struct` from in front of your struct declarations.

If you have a constructor that takes arguments, then you treat the variable being declared as though it were also a function:

```
int gpa = 3.4;
Student aStudent(gpa);
```

Pointers to objects work just like pointers to values of other types. When declaring a pointer to an object, the constructor will not be invoked because the memory for the object has not yet been allocated (using `new`):

```
Student *aStudent = NULL;
```

Note the difference between declaring an instance of an object and declaring a pointer to a (potential) object. When an instance is declared, memory is allocated for the object and the constructor is called. In the second case, a pointer to an object is created, but no memory is allocated (and, consequently, no constructor is called).

3 Memory Management

In Scheme, memory management was taken care of for you. This meant that you could focus on the problem you were solving, rather than on allocating the resources to solve the problem. In C++, you will have to use pointers and memory allocation to dynamically make resources (memory) available to your program. Unfortunately, once you start allocating memory, you become responsible for returning it to the system.

3.1 Pointer Review

Pointers store the memory addresses of other variables. Pointers are declared with the type of the variable to be pointed to and an additional asterisk `*` as a prefix:

```
int *aPointer;
```

Pointers do not start out initialized—you may wish to initialize them to `NULL` to help catch the use of uninitialized pointers. To set a pointer, you can either allocate memory or assign the pointer to point to memory that has already been allocated. The simplest way to set a pointer is to have it point to the memory address of another variable by using the address-of operator, `&`

```
int *aPointer;
int anInt = 5;

aPointer = &anInt;
```

To recover the value stored in the variable a pointer points to, we dereference the pointer using an asterisk (yes, the same symbol used to declare them):

```
cout << *aPointer; // prints 5
```

3.2 new

`new` is a C++ keyword that takes a type and returns a pointer to that type. If you are allocating memory for a class, it will also invoke the class's constructor.

For instance, in C, how would you allocate memory for an integer:

```
int *x = malloc(sizeof(*x));
```

In C++, you can write (and, in fact, we expect you to write)

```
int *x = new int;
```

When invoking `new` to create objects of a class, you give the arguments to the constructor. For instance,

```
Object *foo = new Object(arg1, arg2, arg, arrr);
```

3.3 delete

If C has a positive attitude about memory—let it run **free!**—C++ is a bit more authoritarian. Once you're done with allocated memory, you **delete** it. This brings out a subtle difference in the way C++ thinks about memory. In C, you **free** memory; in C++, you **delete** a(n object through its) pointer.

```
delete aPointer;
```

You may find it helpful to set pointers to `NULL` once they have been **delete**'d so that you don't accidentally reuse the memory that has already been returned to the system.

Note that *every* call to `new` will need a corresponding call to `delete` at some point, or you will have a memory leak.

3.4 Debugging Memory Problems

If you haven't already, learn to love `gdb` or its graphical cousin `ddd`. If you don't, you will either be unhappy or very lucky. You might also want to check out `valgrind`, a memory profiling tool installed on the Nice servers.

3.5 Pitfalls

3.5.1 Out of Memory

When `malloc` fails, you get a `NULL` pointer. Fortunately, in C++, when `new` fails, an exception is thrown, which will effectively kill your program. For assignment five, this behavior is acceptable.

3.5.2 Memory Leaks (Lost Memory)

Memory leaks occur when a piece of memory that has been allocated is never deallocated. This tends to be a big problem when programs get larger, and if you use up too much memory, `new` will fail and your program will probably die.

A common pitfall is to forget to `delete` a pointer. When you allocate memory for a pointer, know exactly where you will `delete` that pointer.

Unfortunately, memory leaks can sometimes be hidden. For instance, never overwrite a memory address if you don't have a copy somewhere else:

```
int *aPointer = new int;
aPointer = new int; // Oops, what happened to aPointer?

int *aPointer = new int;
int *backupPointer = aPointer;
aPointer = new int; // Okay, now we have pointers to both memory
                    addresses
```

3.5.3 Dangling Pointers

Another problem you may run into is that if you `delete` a piece of memory and have another pointer to that same piece of memory, then there's no way for the other pointer to tell if the memory is still valid. One way to help catch this situation is to `NULL` or otherwise reset the members of an object in its destructor. If you follow a pointer to an object that has been deleted through another pointer, you will have an easier time detecting the problem if you have "invalidated" the object.

3.5.4 `NULL` pointers

Don't dereference a `NULL` pointer. If you don't know if a pointer is `NULL`, check it or use `assert()` if you believe the pointer should never be `NULL`. You will lose points if you do not do this.

4 I/O

4.1 Output

To display text to the screen, you use the `cout` object (pronounced 'see out'), which has the somewhat strange syntax of

```
cout << [stuff to output] << [more stuff to output] << ...;
```

Note that you can string together different types of data without having to specify a type. For instance, to output an integer, `x`, and a string, you could do this:

```
cout << "The number is " << x;
```

To include a newline, you can either include `\n` inside a string, or use `endl`, which will also flush the output buffer.

```
cout << "The number is " << x << endl;
```

To send output to `stderr`, you can use the `cerr` object instead of the `cout` object. (This is equivalent to using `fprintf(stderr, ...)` in C.)

4.2 Input

To read input from the standard input, `stdin`, you can use the `cin` object and simply reverse the direction of the arrows:

```
cin >> in_variable;
```

To use `cin`, `cerr`, or `cout`, you must `#include` the `iostream` header (no `.h`) and add the line “`using namespace std;`” to your source code file. Otherwise, you will need to qualify the names appropriately (e.g. `std::cout`).

5 Debugging

Unlike Scheme, with C++ we do not have an interpreter to interact with when a program mysteriously breaks. However, a program called a *debugger* exists which replicates much of this functionality: viewing a backtrace of the program’s function call stack, manually invoking code, and so on.

A popular debugger for UNIX-based systems is the GNU Project Debugger, `gdb`. There are two fundamental strategies for using GDB to help fix your code: postmortem analysis and interactive experimentation.

Postmortem analysis is what you need when your program stops with the infamous **Segmentation fault (core dumped)**. The “core” that gets “dumped” is a copy of your program’s entire memory space at the time of failure. GDB can be used to pick this apart to show you where your program went wrong. Start GDB with `gdb <executable file> <core file>` (replacing the bracketed text with appropriate file names). The most useful command to give GDB after that is `where`, which tells you exactly where your program stopped by printing a “backtrace” of the call stack. You may also use `p <variable>` (“print” variable) to view the values of variables. It will look something like this:

```
#0 0x0804a7e7 in do_if () at commands.c:95
95      *pint=0;
(gdb) where
#0 0x0804a7e7 in do_if () at commands.c:95
#1 0x0804b5dd in call_builtin (kw=0x804d0dc) at execute.c:242
#2 0x0804b43c in run_statement () at execute.c:165
#3 0x0804b49f in run_current () at execute.c:187
#4 0x0804b57f in run_program (first=0x80738f8) at execute.c:225
#5 0x0804a8b2 in do_run () at commands.c:121
#6 0x0804b5dd in call_builtin (kw=0x804d794) at execute.c:242
#7 0x0804b43c in run_statement () at execute.c:165
#8 0x0804b537 in run_direct (code=0x8073cb8 " ") at execute.c:214
#9 0x08048ea3 in main () at basic.c:91
(gdb) p pint
$1 = (int *) 0x0
```

Here GDB revealed that I tried to dereference a NULL pointer. Another useful command in this case is `f <number>` (standing for “frame”) which lets you switch what stack frame you are looking at. The frame numbers are listed in the backtrace above. It is important to explore backwards in the call history, since the actual bug might be in a previous function. For example, a function might have passed a NULL pointer to a function that wasn’t expecting one, leading to a segfault in the called function even though the caller is to blame.

Often you’re not lucky enough to get a core file; your program simply produces the wrong result. In such a case the interactive experimentation strategy is warranted. To do this, you actually run your program within GDB, using `gdb <executable name>`. Starting GDB in this way leaves you with nothing but a command prompt initially. To run your program, use the `r` command (“run”). This is different from running your program on the command line in two ways: first, if your program segfaults you will end up as if you had loaded the core file into GDB as described above, and second, if you hit `Ctrl-C` (break) at any point, GDB will show you where your program is currently executing. If you suspect your code contains an infinite loop, you can hit `Ctrl-C` at some point during the possibly-infinite looping to discover a line of code in the loop (note that though the loop might involve many lines of code, possibly spread over different functions, GDB can only show you which line is executing currently).

Whenever your program is stopped while running (as opposed to after having crashed) you may invoke your own functions interactively. To do this, use the `p` command (“print”), which is able to evaluate an arbitrary C++ expression whose result you want to display.

To solve the most inscrutable algorithmic problems (where the symptoms are less obvious than crashing or infinite looping) you will have to set up experiments using *breakpoints*. A breakpoint asks GDB to stop your program when it reaches a specific piece of code, allowing you to take control. Load your program into GDB and use the `b` command (“break”). Note that if you used GDB to run your program with `run`, you may need to stop it with `Ctrl-C` to return to the prompt to be able to use this command. The `b` command comes in two flavors: one lets you stop at a specific line, the other lets you stop at a function. They look like `b file.cc:92` and `b MyClass::method`, respectively.

When your program stops at a breakpoint (or `Ctrl-C`), you can manually control its execution using `n` and `s`. `n` runs until the next line of the current function, whereas `s` will move into any functions that are called, letting you step through their code as well.

Here is a summary of the most useful GDB commands:

Command	Description
<code>r <arguments></code>	run program with <code>jargumentsj</code> as command line args
<code>Ctrl-C</code>	break
<code>n</code>	next line
<code>s</code>	next line, stepping into called functions
<code>b file:line</code>	set line breakpoint
<code>b methodName</code>	set function breakpoint
<code>d <number></code>	remove your <code>n</code> th breakpoint
<code>p <expression></code>	print the value of an expression
<code>display <expression></code>	print after each step
<code>where</code>	show stack trace
<code>f <number></code>	switch to another stack frame
<code>trace methodName</code>	automatically display each invocation of a function

6 Other C differences and style

- Use the `const` keyword with a declaration instead of `#define`.
- C++ has a proper `string` type, which you should use in preference to `char*`.

- In C++, it is traditional to use camelCase instead of names_with_underscores.
- Use the // single line comment character
- Use appropriate spacing around infix operators. Use parentheses to clarify order of operations.
- Continue to use emacs or vi to indent code for you.

7 Example: moving a wordprocessor from C to C++

You've just been hired by Mom's Old Fashioned Software Company to help update their classic wordprocessing product, WordFlayer. The company founder, Mom, wrote the original version herself in 6502 assembly language, working in her parents' garage. Several years later, the program was rewritten in C after a rat chewed through the 5.25" disk that held the only copy of the source code. To keep up with the times, Mom wants WordFlayer rewritten yet again, this time in C++. Part of the motivation for this project is the large rat hungrily eyeing the 5.25" disk that holds the only copy of Mom's C Compiler, but at the same time the company wants to take advantage of C++ to make their software more flexible and easier to maintain, so they can add lots of features.

Below are declarations for WordFlayer's most important C data structures, along with prototypes of some functions that manipulate them. Your job will be to declare analogous C++ classes.

WordFlayer documents are currently implemented as linked lists of paragraphs (pieces of styled text).

```
typedef enum {
    STYLE_PLAIN,
    STYLE_BOLD,
    STYLE_ITALIC
} textstyle_t;

typedef struct {
    char *text;
    textstyle_t style;
    int fontSize;
    char *fontName;
} paragraph_t;

typedef struct _llist_t {
    void *element;
    struct _llist_t *next;
} llist_t;

typedef struct {
    char *fileName;
    int cursorPos;
    int curPage;
    int nPages;
    llist_t *els;
} document_t;

/* create a document */
document_t *newDocument(char *fileName);

/* create and add document elements */
```

```

paragraph_t *newParagraph(size_t bufferSize);
void addParagraph(document_t *d, paragraph_t *p);
void removeElement(document_t *d, void *el);

/* calculate the size of paragraphs */
int paragraphWidth(paragraph_t *p);
int paragraphHeight(paragraph_t *p);

/* given a cursor position, find the paragraph containing it */
paragraph_t *posToParagraph(document_t *d, int pos);

/* given the first element on some page and the height of a page,
   return the element that should be first on the next page */
llist_t *nextPageStart(llist_t *el, int pageHeight);

/* merge two adjacent paragraphs into a single paragraph */
paragraph_t *mergeParagraphs(paragraph_t *p1, paragraph_t *p2);

```

Let's redesign this package in C++. We need to create two new abstractions: Documents and Paragraphs.

For each abstraction, define a constructor and any necessary selectors. Then add the state fields that you'll need to support the constructor and selectors, using appropriate C++ types. Next, add methods that provide the same functionality as the existing C API.