Main topics of the week:
- Programming Style
- Preprocessor Directives
- Arrays, Pointers, References
- Heap allocation
- Function Pointers
- Constant Objects

Programming Style

The nature of software is that it evolves over time. A program that works tends to be reused, modified, and otherwise mutated. So it is important to keep this in mind when writing any code. It is very unlikely that you would write some code once and it would never be looked at again. It is far more likely that you will look at your program many times in the future, and from different perspectives, and that others will look at your program. In large software projects, many engineers may work together on pieces of software, constantly examining and having to understand each others’ code.

Adopting a good programming style is as important as getting the code to work correctly in the first place. The goals of a good style are to promote readability, maintainability, and extendability. Aspects of good style range from basic formatting issues (consistent and clarifying use of indentation and blocking) to meaningful name selection, simplicity of code, compactness of code, elegance of code, and so on.

We know that formatting means nothing to the compiler – it ignores all white space except as needed to separate tokens. The compiler could easily tolerate a large program written on a single line with a minimal number of space characters. However, we humans need the visual information that good program organization conveys. White space can be very effective for communicating the structure of a program by using consistent indentation. We know that the indentation does not affect the control flow, but we can use indentation to show the control flow. We can use spacing to visually group the pieces of a complex expression, and this can reduce the number of parentheses, which can be distracting. For example, double blanks can be used to separate sub expressions that are combined with logical conjunction or disjunction. The point is to use formatting consistently and effectively to aid in understanding a program.

Good programming style will also use comments effectively. It is important to find the right balance between too little and too much commenting. Most code deserves some level of commenting, but comments that repeat the obvious detract from program comprehension. Comments should add information, not repeat it. Remember that comments are not compiled, so it is important to keep the comments in sync with the code as the code evolves.

The visual style of comments can be important to making code understandable. Multi-line comments begin with /* and end with */, but it is a good idea to use some visual clue to help differentiate between block comment text and code, e.g.,

/* This is an example
 * of a multi-line comment with
 * a clear visual clue that each
 * line is a comment.
 */
Remember that multi-line comments do not nest. Using only the // style comment, even for multi-line comments, not only provides a good visual clue about what is a comment, but also means that the /* and */ can be used for commenting out whole blocks of code during debugging without running afoul of the lack of comment nesting.

Names of variables and functions can convey a great deal of information, so it is worth investing time to choose names carefully. Certainly some simple variable names like i and j are appropriate for local used loop indices, but most variable names should indicate the purpose of the variable. A comment at the declaration can provide useful clarification as well. The case sensitivity of C/C++ allows case distinction to be used for making phrases into variable and method names.

Other style issues in writing software include practices relating to variable use. Variables should not be used for multiple purposes when that use can lead to confusion. C++ allows variables to be declared at any point, so a variable’s declaration can be placed at the point of its first use. Generally, the scope of a variable should be as restrictive as possible to prevent it from being used where it wasn’t intended.

The program structure is an important aspect of style. Code that is too complex can obscure the logic of the program. But code that breaks expressions and statements into too many small pieces can also be hard to understand because of the large number of statements. C and C++ are notorious for compact expressions that can pack a lot of logic into a single expression. This can be a good feature for maintainability by keeping together actions that logically belong together. Compact notation should be used in a thoughtful way, always keeping in mind design and maintenance reasons for using the compact notation, rather than just compressing expressions for the challenge of making them as short as possible.

The general idea to keep in mind is that good style can be an effective tool for expressing and supporting design intent.

Preprocessor Directives
The first thing done by the C/C++ compiler is to run the C preprocessor (which you can actually run as a separate tool named cpp). The preprocessor reads the input and writes it back to output, making some transformations as directed. It removes comments from the source stream and may change the spacing. The directives to the preprocessor are lines that begin with a # character as the first character on the line – this is the only place where column position has any significance in C/C++ source.

The most common directive is #include. This is followed by a file name enclosed either in double quote marks or angle brackets (the less than and greater than signs), e.g., #include <iostream>. This directive means that the preprocessor will fetch the named file and insert its contents into the input stream. This activity can be nested, i.e., an included file may include another. The preprocessor does keep track of the current file name and line number for properly labeling error messages. (And this can be overridden by the #file and #line directives.) Included files can be named with full path names or relative path names. Relative names will be searched for in a list of standard directories (e.g., /usr/include) and possibly the current directory. Check the manual page for a particular implementation to see the difference in using angle brackets or double quotes. Typically, that distinguishes standard header files packaged with the
compiler from local project specific header files. Command line arguments are also typically available for specifying the path of directories to be searched for files.

Another common directive is symbol definition with `#define`. Prior to C++’s introduction of `const`, this was the standard way of defining a constant value, e.g.,

```
#define HOURS_PER_DAY 24
```

A `const` variable in C++ is now preferable – it conveys type information to the compiler, while the `#define` is a simple textual substitution. However, this means that `#define` can be used for simple code fragments. Moreover, `#define`’d symbols can be used to control compilation with the conditional compilation directives: `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`. These directives can be used on symbols (whether they are defined or not) or with simple expressions, doing string comparisons or arithmetic comparisons. Such expressions are evaluated at compile time. The effect of the directives is to determine whether to ignore enclosed code. This gives a technique for including platform or feature specific code in common source that is compiled with different symbols defined by the compile command (or using built-in platform specific symbols that refer to machine architecture or compiler version). A common use is to prevent a header file from being included more than once:

```
#ifndef _THIS_HEADER
#define _THIS_HEADER
    ...contents of the file...
#endif
```

The `#define` directive may also be used to define a macro that has positional arguments. This looks like a function definition, but has no type information, and is again mindless textual substitution. Because it is simple substitution, one must be careful about inadvertent grouping problems. For example,

```
#define SQUARE(x) x*x
```

seems to be a reasonable definition of a macro that squares a value. In the use:

```
n = SQUARE(8)
```

the preprocessor certainly does the right thing, resulting in `8*8` in the output that is fed to the compiler. However,

```
SQUARE(i+1)
```

becomes

```
i+1*i+1
```

with the mindless substitution, and this clearly groups in a way that gives the wrong result. The moral is that parentheses should be used liberally in macro definitions to avoid such improper grouping.

---

**Arrays and Pointers**

One of the most common stumbling blocks in learning C is understanding pointers and their relation to arrays and other variables. C has variables, and all variables must be declared. The syntax of the use of a variable is to just mention the variable name in an expression. That is effectively a dereference of the variable, i.e., it means to fetch the variable’s value. We can imagine that the compiler associates the variable name with a memory address where the value of the variable is stored. The compiler also knows the variable’s type, so when it sees a use of the variable, it can (based on the type) fetch the right number of bytes beginning at the address, and interpret the bytes properly, again according to the type. If the variable is used as an lvalue (e.g., on the left side of an assignment) then the process is reversed as a new value is stored rather than fetched.
So where do pointers come in? They are just variable types that provide an additional level of indirection. From the discussion of variable access and storage above, we can see that memory addresses are used internally – pointer variables allow us to access and manipulate the addresses themselves, and memory addresses become another type of value like integer or floating point. By declaring a variable to be a pointer (and we always have to say what type we are pointing to) we mean that the value being stored is an address. The pointer variable itself is associated with an address (as are all variables), and at that location we are storing some bytes which are interpreted as the address of the data we really care about. With a pointer declaration, we only are arranging for space to store an address, so the question arises, what address should be stored there? We need a valid memory address, and one way to get such an address is to simply take the address (using the & operator) of another variable (of the right type). When we use a pointer variable in an expression, its value is an address. Most likely, we are really interested in the data at that address, so we use the “contents of” operator (the symbol *) to fetch the contents from a pointer value. It might help to think of this in Java terms – an object variable in Java just declares a “handle” for an object. We must always get an object from somewhere (using new or another object) to assign to the handle. Effectively, in Java all non primitive variable types are pointers, and assignment between such variables just means pointer assignment. The only difference is that in Java, you cannot see the address – it’s as if the “contents of” operator is always applied when the object variable is mentioned in code.

C (and C++) allows us to distinguish the manipulation of the pointers from the data pointed to. One powerful place where this comes into play is with arrays. An array declaration in C always specifies a size (possibly calculated from an initialization list), and informs the compiler to arrange for that much contiguous space (on the stack or heap). The actual type of the array variable itself is that it is really a pointer to the space. The simple variable name is then treated as a pointer type, whose value is constant – we can’t rebind the array to a different location. However, the storage space is of course changeable, so we can use the array name, along with the array operator, to index the values in the array with read or write access. So the array name is then the beginning address of the array, i.e., the address of the first of the array elements. This is where pointers come in. To traverse the array, we can start with a pointer initialized to the array and then keep incrementing the pointer to step through the array. Remember that C requires pointers to specify the type pointed to, and this means the arithmetic with pointers can be scaled properly. E.g., if we have a pointer to an int, then incrementing the pointer means to increase the byte address by the size of an int. This results in much more efficient access of the array elements since the address arithmetic to find the element’s location as an offset from the beginning is not performed repeatedly – by using a pointer we are effectively keeping a cursor to the desired memory location.

It may seem like pointers and arrays are interchangeable, and in many situations we don’t care whether the type is array or pointer. Where the difference is important is when it comes down to allocating space. A pointer variable declaration only gets us space to store an address. An array declaration gets us a bunch of contiguous space for values. Often we will use both arrays and pointers. The main thing to keep in mind with pointers is to understand where the space comes from. Is it dynamically allocated on the heap with
malloc or new, or space on the stack corresponding to an array declaration, or a global variable?

```c
int i;    // Single int value
int a[10]; // Storage for 10 int values
i = a[0];  // Assignment between ints
a[i] = a[i+1];  // Assignment between ints (i better be in the range of the array!)
int *p;    // Storage for the address of an integer
p = & i;   // p points to where i is stored
p = & a[i]; // p points to the storage of i’th element of the array
p = a;     // Same as p = & a[0]
++p;       // Now p points to a[1]
*(a + i);  // Same as a[i]
```

**Heap Allocation**

The standard C library provides functions for the allocation and de-allocation of dynamic memory in the process image for the executing program. This is often called the heap memory. The library function `malloc` is given a single argument, which is the number of bytes to allocate, and returns the address of those bytes (or the NULL pointer if allocation didn’t succeed). The address is a generic byte pointer, and should be cast to be a pointer of the appropriate type. For example, if we have an integer variable `nvalues` that is the number of double precision floating point numbers we want to allocate, the fragment of code to do this would be:

```c
double *valueList = (double *) malloc(nvalues * sizeof double);
for (int i = 0; i < nvalues; ++i)
    . . . valueList[i] . . .
```

Notice that `valueList` is declared as a pointer to double and is set to the address of the dynamically allocated memory on the heap. After this, `valueList` can be used as if it were declared as an array of `nvalues` double’s. This is because of the basic storage equation that defines the equivalence of array and pointer notation:

```
    a [ i ]    is equivalent to     *(a + i)
```

In a well behaved program, dynamically allocated memory should be freed when it is no longer needed. The library function `free` is given a single argument which is a pointer to previously allocated memory. Once freed, the address should not be used since it may be re-allocated by a subsequent call to `malloc`.

```c
free(valueList);  // We’re done with this chunk on the heap
```

**Pointers and Function Call Protocol**

The calling convention used in C programs is call-by-value. That is, a function call expression means to evaluate the arguments, and then copy the values into an activation frame, then jump to the function’s code, where it can access the parameter values from the activation frame. This means that we have call by value and the caller’s data is not affected by the function’s code. However, once we have pointer types, the value passed to a function could be an address, and the function could use that address to affect values stored at the location of the address. This means we can achieve the affect of call by reference in C by passing address values, and knowing in the function to access the
contents at the passed addresses. Syntactically, it’s messy and often confusing to get the &’s and *’s in the right places, but it certainly can be done. A simple swap function can illustrate this idea as shown in the example below.

References in C++

C++ introduced the idea of the reference type. The motivation for this type was to ease the definition of overloaded operators, but it turns out to be a really nice idea for general use. To say that a variable is a reference in C++ (using the & operator in its declaration), we mean that it is just another name for some other variable, i.e., an alias. It’s like a pointer in that we aren’t declaring storage for a value, but in the case of a reference we aren’t even declaring changeable storage for an address – the compiler is taking care of binding the symbol to the storage of another symbol. Thus references cannot be changed once initialized – we simply can’t get at the binding. However, assignment to a reference variable is meaningful – the reference is just an alias for another variable, so assignment is just like assignment would be using that other variable’s name. Used just as aliases, references would not be very interesting, but as function arguments they are very powerful. In effect, they become aliases for the caller’s variables, and so permit true call by reference parameter passing. Syntactically, all that is required is the addition of ‘&’ on the formal parameter to a method. The compiler arranges transparently for everything to be call by reference for that parameter. Moreover, a function can return by reference as well. When references are combined with const (to give references to constant values), we get the efficiency of passing pointers, but also the safety and syntactic simplicity of call by value.

The following example shows call by value, call by explicit pointer value, and call by reference:

```cpp
#include <iostream>

void swapv(int x, int y) { // Call by value
    int temp = x; x = y; y = temp;
}

void swapp(int *px, int *py) { // Call by pointer
    int temp = *px; *px = *py; *py = temp;
}

void swapr(int & x, int & y) { // Call by reference
    int temp = x; x = y; y = temp;
}

int main() {
    int a = 13, b = 7;
    int *pa = &a, *pb = &b;
    cout << "a=" << a << " b=" << b << endl;
    cout << "Calling swapv" << endl;
    swapv(a, b);
    cout << "a=" << a << " b=" << b << endl;
    cout << "Calling swapp" << endl;
    swapp(&a, &b);
    cout << "a=" << a << " b=" << b << endl;
    cout << "Calling swapr" << endl;
    swapr(a, b);
    cout << "a=" << a << " b=" << b << endl;
}
```

Atkins – Spring 2010  -  6 -  CIS 330 C/C++ and Unix
Function Pointers

In addition to pointers to data types, C and C++ permit pointers to functions. In fact, a function name by itself is considered by the compiler to be the “address” of the function (much as an array name is the address of the first element of the array). The way the compiler implements a function pointer (maybe as an address in the instruction space) is not so important, but rather what we can do with the function pointer. A function pointer variable can be used to call the function, since that’s what we do with functions. That is, when we dereference a data pointer variable, we get the value of the data; when we dereference a function pointer, we get the function, and the reasonable thing to do with the function is to call it. The syntax of declaring and using a function pointer looks like:

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

char * reverse(char * s) {
    // Reverse the string
    const n = strlen(s);
    for (int i = 0; i < n/2; ++i) {
        char temp = s[i];
        s[i] = s[n - i - 1];
        s[n - i - 1] = temp;
    }
    return s;
}

cchar * shift(char * s) {
    // Shift the string to the left
    const n = strlen(s);
    char first = s[0];
    for (int i = 1; i < n; ++i)
        s[i-1] = s[i];
    s[n-1] = first;
    return s;
}

int main() {
    char test[] = "Hello, world";
    char * (*fp) (char *); // fp is a pointer to a function
    fp = reverse;
    cout << (*fp)(test) << endl;
    fp = shift; taking, returning a string
    cout << (*fp)(test) << endl;
    return 0;
}
```

Constant Objects

C++ introduced the idea of a read only data location and this allows us to define constant objects. The keyword `const` is used in a declaration to indicate to the compiler that the value of the declared object should not be modified. From the compiler’s point of view, this means that the object cannot be an lvalue, i.e., on the left hand side of an assignment expression. The compiler implementation may choose to use some form of read-only storage on the platform that could provide hardware protection against modification and/or more efficient access. But from the language point of view, const is a way of expressing design intent and getting the compiler to enforce that intent consistently.

With pointers, the compiler must be extra careful not to lose track of the constant-ness of an object. That is, if the address of a const variable is passed to a function, then the function must declare the pointer as a pointer to a const object – otherwise there
would be a gaping hole in the protection afforded by const. It is possible for a pointer to be const, which means it cannot be assigned to point to another object. So we have the possibility of the object being constant, the pointer being constant, neither, or both. It is not an error to assign the address of a non-constant object to a pointer to a constant object since we are not opening up access that is not intended, but as described above, assigning the address of a constant object to a pointer to a non-constant opens a hole, and the compiler will not permit that:

```c
const int n = 15;    // constant integer n – must be initialized
const int m = 15;    // non constant integer m
n = 17;    // Error – attempt to change constant
m = 17;    // OK – assignment to normal non constant
int *pn = &n;    // Error – access through pointer loses constantness
int const *pn = &n; // Still an error – pointer is constant, data is not
int const * const pn = &n; // OK – constantness is preserved by pointer
int const * const pn = &n; // Also OK – const of n preserved, pn const too
int *pm = &m;    // OK – no constantness
int const *pm = &m; // OK – m can’t be changed via pm
```