

Imperative Programming

Imperative programming is about writing *statements* that perform some *action* that changes state, thereby effecting computation. An imperative program typically consists of the following elements:

- data type definitions (e.g., builtin types and user-defined types)
- typed variable declarations (global and local variables and parameter lists)
- expressions and assignment statements (arithmetic and relational expressions)
- structured control flow statements (conditionals and loops)
- lexical scopes and blocks (providing *locality of reference*)
- procedure/function declarations and definitions (parameterized blocks)

Data type definitions allow a programmer to define new types using builtin or previously defined types. For example, the following C/C++ type definition defines a structure (or record) `TreeNode`, which consists of an integer type followed by two “pointers” to `TreeNodes`.

```
typedef struct TreeNode {
    int x;
    TreeNode *front, *back; // front and back pointers to use in a doubly-linked list
};
```

Typed variable declarations are very important as they allow the programmer to specify the range of values that a given variable may assume during the execution of a program. For example:

```
int x = 0; // defines a variable x of type integer, whose initial value is zero.
```

Some languages guarantee that variables declared as a builtin type are given a default value automatically. For example, Java ensures that any integer type declared without explicit initialization is initialized to zero. C/C++ on the other hand does not perform any default initialization. The value of an uninitialized variable is undefined.

When a variable is declared, space for the variable can be determined as the sum of the space required to represent the type. For example in C/C++ on most 32-bit machines, `sizeof(char)` is 1 byte, `sizeof(short)` is 2 bytes, `sizeof(int)` is 4 bytes, and the `sizeof` of a pointer is 4 bytes. **So, the `sizeof(TreeNode)` is.....what?**

Assignment Statement

The use of the assignment statement is a distinguishing feature of an imperative programming language. The assignment statement is how state is changed in a controlled fashion.

```
int x = 0; // initialization at the point of declaration
...
x = x + 1; // get the r-value of x, add 1, and store the result into the l-value of x
```

This simple assignment statement means that the expression on the RHS is evaluated to produce an *r-value*, and the r-value is then assigned to the storage location to which the variable *x* is bound, called the *l-value*.

- **r-value** means the right value or the stored value
- **l-value** means the left value or location.

Notice that in the above assignment statement, the variable *x* occurs on both the LHS and the RHS. When the variable *x* is declared, *x* we say that it is **bound** to some storage location and becomes the **identifier** for that location. If *x* is used in an expression on the RHS of an assignment operator, then its **value** is the contents of the storage location bound to *x*. So, the **r-value** is the value stored at the location given by the **l-value**.

When a variable is used as the target of an assignment, then the value of that variable is the location to which the variable is bound. So, the l-value of *x* will be some storage address (global, stack, or heap storage) corresponding to the memory location allocated for the variable *x*.

This means that assignment is a destructive operation in that it overwrites the contents of some memory location with the value computed for the expression appearing on the RHS.

An expression that does not have an l-value cannot appear as the target of an assignment operator. For example:

```
1 = x + 1;
```

Is illegal, since 1 is a constant, and constants do not have l-values, but the following is legal if 'a' is an array. Why?

```
a[1] = x + 1; // assign the value of x + 1 to location 1 in array a
```

L-Values and R-Values

You can understand any expression or assignment statement in an imperative statement in terms of the l-values and r-values of every variables used in the expression or assignment statement. Thinking in terms of l-values and r-values also allows you to reason through complex pointer dereferencing and pointer arithmetic. In the examples below, let's assume that the compiler interprets every expression in terms of functions called `rval` and `lval`, which when applied to an argument, return the stored value in case of `rval` and the "address" or location value in the case of `lval`:

1. literal constants only have r-values, but not l-values. For example, `rval(1) = 1`, `lval(1)` is undefined.
2. variables have both r-values and l-values. So the expression: `x = x * y` is interpreted by the compiler as `lval(x) <- rval(x) * rval(y)`
3. variables of pointer type have an r-value that is the l-value of some other variable. In C/C++, two special unary operators are used to override the default action of the compiler which is to compute the l-value for the target of the assignment and the r-value for every variable on the RHS of an assignment. The unary '&' operator (address-of operator) applied to a variable (not a constant since constants have no l-value) overrides the r-value computation and instead returns the l-value. Likewise, the unary '*' (pointer dereference) operator returns the r-value when applied to a pointer variable, which is the l-value of something else. For example:

```
int x = 5; // lval(x) is some (stack) address, rval(x) == 5
int *p = &x // rval(p) == lval(x)
*p = 2 * x; // rval(p) <- rval(2) * rval(x)
// What are the values of P & x at this point?
```

4. Declared functions/procedure have l-values but no r-values.

```
int f(int y); // lval(f) is some global address
typedef int (*IFP)(int); // pointer to an int function that takes an int argument
IFP g = &f; // lval(g) <- lval(f)
(*g)(5); // (rval(g))== lval(f), so *g invokes f with argument rval(5)
// the function call operator () has higher precedence than * so
// we have to write (*g)(5) to dereference g to invoke f(5)
```

Structured Control Flow

Control flow in an imperative language is most often designed to be *sequential*, meaning that each statement is executed in the order in which it appears statically in the written program. Sequential composition of statements is the normal style of programming. However, it is possible to execute statements *concurrently*, independent of one another, except for the occasional sharing of data, if a language supports concurrent execution (e.g., Java).

A program is called *structured* if the flow of control through the program is evident from the syntactic (static) structure of the program text. **The purpose of structured programming is to allow the programmer to be able to reason about the dynamic execution of a program by just analyzing the program text.**

Structured programming was invented as a technique for eliminating some of the complexity that arises when programs become large and the difficulty of understanding the sequence of actions take by the program increases. This complexity is the source of logic errors resulting in run-time “bugs”. Common “patterns” of control flow were used over and over by early programmers, and so they designed high-level languages that encompass common patterns so that programming effort is reduced.

Examples are:

- iteration statements
- selection statements
- procedure/functions calls

Iteration statements (loops) such as **while**, **do-while (repeat-until)** and **for**, provide for structured iteration without the use of goto statements. Goto’s are considered to be a low-level control flow operation that should only be used to implement high level structured programming constructs.

Selection statements consist of branching binary conditionals (**if-then-else**) and multi-way conditionals (**case** or **switch**).

A **procedure (or function) call** transfers control (and argument data) to another part of a program in a well-defined manner and control flow is returned to the statement after the point at which the procedure call is made when the procedure executes a **return statement**. The returning context is called a **continuation**.

Iteration Constructs

Different constructs allow the specification of either *definite* or *indefinite* iterations.

A **definite iteration** is executed a fixed number of times. E.g.:

```
for (int i = 0; i < 10; i++) {  
    a[i] = 0; // initialize each array element to zero  
}
```

This is an example of a definite iteration using a **for loop** that iterates through the first 10 elements of an array, initializing each element to zero. (Note: in C/C++, arrays begin at index 0). The number of iterations can be determined prior to execution of the program containing this loop. In this case, the ‘for loop’ is just a compact way of writing 10 sequential assignment statements, all of which are identical except for the value of the array index.

An **indefinite iteration** relies on a dynamically computed value at run-time to determine whether or not the iteration should continue. For example:

```
int m = 0;  
while (n > 0) { // keeps looping as long as n is positive  
    m = m * n;  
    n = n - 1; // will eventually change the state of n to zero and terminate the loop  
}
```

The number of times the loop is executed depends on the value of n at run-time.

QUESTION: How do you know for sure that an indefinite iteration will terminate? E.g., in the loop above, how do you know before you run your program that n will eventually become less than or equal to zero?

Note that the `{ }` braces used in a while loop introduce a new **lexical scope** or **block**. A lexical scope is a textual region of a program in which variables are declared that are “local” to the scope. This means that the meaning of the variable only makes sense within the boundaries of the lexical scope or block in which it is declared. Said another way, the **extent** or **visibility** of the variable is the entire textual block and the **lifetime** of the storage location associated with the variable is tied to the lifetime of the block (the storage location is typically the run-time stack).

Iteration Constructs

The common iteration constructs in C/C++ are:

```
while (condition) statement;
while (condition) { statement; statement; ...; };
```

The while loop evaluates the condition and if true, the body of the loop is executed. The body is either a single statement or a block containing one or more statements. At the end of each iteration through the loop, control transfers back to the top of the loop to evaluate the condition, and the body of the while loop may be executed again. The iteration will only terminate if the body of the while loop eventually causes a state change such that the condition evaluates to false.

The do-while loop construct (similar to Pascal's repeat-until) moves the condition check to after the body of the loop:

```
do statement while (condition);
do { statement; statement; ...; } while (condition)
```

The do-while iteration unconditionally executes its body once, then evaluates the expression in the while clause. If the expression evaluates to true, the loop is repeated; otherwise it terminates.

```
for (<initialize>; <test>; <step>) <statement>
for (<intialize>; <test>; <step>) { <statement-list> }
```

Note that a for loop is a more restricted form of a while loop, where initialization occurs just before the while and the step (usually) has an impact on the evaluation of the test:

```
<intialize>
while (<test>) {
    ...
    <step>
}
```

Look at the C grammar file handed out in class and find the BNF rules that describe the iterations statements.

Special Iteration Constructs

As a special case, note that the following for loop is a “forever” loop, equivalent to a non-terminating while loop:

```
for (;;) { ... } /* loop forever */
while (true) { ... } /* loop while true, i.e., forever! */
do { int x; stmt; } while(false) /* execute statement once, but in a new lexical block */
```

Most of the time iteration constructs are *single-entry, single-exit*. However, it is sometimes necessary to be able to terminate a loop prematurely if some special condition occurs that necessitates exiting the loop without the controlling condition being satisfied. For example, the following C/C++ code fragment illustrates “breaking” out of while loop:

```
int y; // y is in the “outer” scope
...
while (cond == true) {
    int x; // x is local to the while blocks scope (its extent and lifetime)
    ...
    if (x < y) { // special case...
        break; // leave while loop
    }
    ... // normal case
}
```

The `break` statement causes control to leave the loop and transfer to the point at the end of the enclosing loop. In the case of a nested loop, control transfers out of the enclosing block into the outer block, e.g.:

```
while (cond1 == true) {
    while (cond2 == true) {
        if (x < y) // special case
            break; // leave inner loop, but not outer loop
        ...
    }
    ... // control resumes here after a break from the inner loop
}
```

Special Iteration Constructs

It is also sometimes necessary to force a loop to be re-entered from the “top” before the loop has reached the “bottom”

For example:

```
while (cond-expr == true) {
    ... // do something while cond is true
    if (a == b) {
        ... // do something special
        continue; // transfer to start of while and re-evaluate cond
    }
    ... // remaining statements of while loop
}
```

Notice that normal looping constructs, along with `break` and `continue` are just a more structured way of programming common `goto` control flow. The “head” of a loop (`while`, `repeat`, `for`) is implemented as a labeled statement. The “tail” of the loop an **unconditional goto** statement with the label of the head as the target of the `goto`. Depending on the type of iteration construct we are dealing with, evaluation of the conditional expressions controlling the definite/indefinite iterations determine whether or not the loop will continue. The `while` and `do` loop constructs are programmed using conditional and unconditional `goto` statements:

```
start: // label head at start of loop
    if (cond-expr == false) goto end;
    ... // body of loop
    goto start; // unconditional branch
end:   ... // continue with statements following while loop
```

A `do-while` (`cond`) loop is terminated by a **conditional goto**:

```
start:
    ... // body of loop
    if (cond-expr == true) goto start;

    ... // continue outside of loop when cond-expr is false
```

Procedures and Functions

A **procedure** is a named scope that is parameterized. That is, a procedure is a language construct for naming a scope containing local variable type declarations and statements, collectively called the **procedure body**, with the ability to pass argument values into the scope. The **formal parameters** of a procedure allow additional *values*, *variable references*, or *names* to be bound into the scope, depending on the calling convention semantics defined by a particular programming language for passing arguments to procedures.

The technical difference between a procedure and a function is that a procedure may produce a visible side effect, whereas a function will not. A side effect is a visible change of state of some data value not defined within the body of the procedure definition. A function on the other hand should be *pure*, in that it only changes its local state in mapping its input(s) values into some output without having a visible side effect outside of the function. In addition, a procedure is not required to produce a return value since it changes state by causing some side effect. A function should have at least one argument, a procedure may have zero arguments.

In some languages, like C and C++, the terms **function** and **procedure** are often used interchangeably, but a distinction should really be made. For example, we often say “this is a function declaration or function prototype” without regard to the fact that perhaps the “function declaration” being defined is really for a side-effecting procedure.

```
int error; // a global 'error' variable declared in file scope
void checkfile() { // procedure returning no result, but causing a side effect
    ...
    error = open("/etc/passwd"); // open Unix password file
    ...
}

int fact (int n) { // function mapping n to n!, with no visible side effect
    assert(n >= 0); // always test the precondition
    return (n == 0) ? 1 : n * fact(n-1);
}
```

We refer to global symbols in C/C++ from another file scope use the ‘extern’ qualifier. For example: `extern int fact(int n);`

Procedures and Functions

Actual procedure/function definition syntax differs from language to language even among **strongly typed** imperative languages, but they have similar syntactic structure.

- A procedure may or may not have a return type (e.g., it may be implicitly **void**). A function always has a return type (e.g., it may be implicitly **int**). In both cases, parameters should always have their types explicitly declared.
- Procedures and functions are named, but the name may be overloaded to have different meaning depending on the type of the arguments. (e.g., binary operator '+' is overloaded depending on the type of the operands: int, float, string, etc.)
- a procedure may take no arguments, a function should have at least one.

The **type of a function** is well defined. For example, the factorial function maps a natural number to a natural number:

fact: nat->nat

A **type signature** constrains result values and the argument values that can be passed to a function to values within the specified **type domains**. A function **prototype** or **declaration** defines the procedure/function name, parameter names and types, and the return type. The **definition** defines the body. Note that the domain of the factorial function is actually larger than it needs to be. The factorial function should be statically restricted to argument values drawn from the set of non-negative integers. However, its domain is the entire set of integers, so we use a precondition to restrict the value of the argument at run-time (dynamically), instead of at compile-time (statically).

It would be nice if we could statically declare the domain such that it is restricted to the values in the range 0..MAXINT, and have the compiler check this for us. Instead, we are forced to put a run-time check into our function body, that checks to make sure that $n > 0$. However, we can call the fact function with a negative value:

```
int n = x - y; // assume x < y
int result = fact(n); // n is negative at this point
```

Questions: If we do not check the argument value at run-time, what is the result of passing a negative value to the fact function? Why can't we restrict the argument domain at compile-time in C or C++?

Procedures and Functions

To summarize:

- Functions return values by mapping one or more argument values to an an output value of some type. So functions can be used in expressions or conditionals:

```
int result = (a * b) + fact(n); // what is the order of evaluation here?
if (fact(n) > 1)
    ...
```

- Procedures cause state changes only. They should not be used in expressions, For example, the following expression is meaningless given the standard definition of the C standard library procedure ‘printf’:

```
int result = (a * b) + printf("hello, world\n");
/* Note that procedures often return an integer result value that denotes the status
   of having executed the procedure. */
```

For example, it is quite common for operating “system calls” (which are mostly procedures, not functions) to return an error code status as a result. By convention, negative integers are used to indicate an error result code. This result value is not really the same as the result value computed by a function, since it usually represents an indicator of success or failure of having caused some side effect, as opposed to a result value obtained by some mapping of the input argument(s) into a result. For example:

```
int open(const char* file, int mode)
{
    if (file == NULL) {
        return -1; // invalid file name

        if (open(file, mode) < 0)
            return -2; // system open failed
        ...
    }
}
```

Note that the open procedure call in C/C++ is used like a function in a conditional expression, because it returns a status result.

Procedure/Function Environments

We can think of a procedure definition as defining a new execution **environment**, that is delimited by a scope that we introduce using whatever scope definition mechanisms exist in our language of choice. Such as, **begin...end** or “{...}”. For example, the following C/C++ function computes the length of a string:

```
int length (const char* string)
{
    int len = 0;  local variable, allocated on the run-time stack for the function

    if (string == NULL || *string == '\0') // note the evaluation order in the conditional
        return 0;

    while (*string != '\0') {
        len++;
        string++;
    }
    return len;
}
```

There are several things worth noting in this example:

- we are defining a function named ‘length’, that given a single argument of type “pointer to character” (i.e., a string of characters), we compute the length as the number of characters in the string and return the length as an integer value.
- a scope is introduced that defines an *environment* containing local variable declarations and the statements required to compute the length of a string
- the formal parameter `string` is a “dummy” name for an argument that will be bound to a run-time stack location when the function is called. When a function is called, its environment is **activated**. The environment consists of **bindings** of local variables to stack locations (l-values) and bindings of argument *values* (r-values) to stack locations associated with named formal parameters. The code body of a function is already bound to the name of the function when the function is first compiled. So at run-time, only the variable bindings to stack locations must occur.

Calling Conventions

When a procedure or function is called, argument values are passed into the scope of the procedure, and bound into the local environment. The local environment is often called an *activation record*, which corresponds more concretely to the idea of a stack activation frame. There are different semantics that can be applied to the way that argument values are bound to formal parameters:

- **Call-by-value.** When an argument is passed to a procedure/function invocation, the r-value is passed on the stack (or possibly in a register). That is, the value stored at the memory location of the argument is *copied* into the stack activation frame of the procedure. In the case of a constant, the value itself is copied. Note that it is not possible for a procedure to change the actual argument since it does not have access to the l-value.

```
int n = 10;
fact(n); // invoke fact with the value 10
fact(5); // invoke fact with a constant
```

- **Call-by-reference.** When an argument is passed to a procedure/function invocation, the l-value is passed on the stack (or possibly in a register). That is, the address of the memory location of the argument is passed as the value of the formal parameter. Since the l-value is passed, it is possible for the procedure/function to update the actual memory contents of the argument, which is defined in the scope of the caller, not the scope of the procedure. So call-by-reference allows a called procedure to change the state of the environment of the calling procedure! So, all arguments must have l-values. A constant argument is given a temporary memory location so that an l-value can be passed.
- **Call-by-value-result** (also called *copy-in/copy-out*). When an argument is passed, its value is copied from the memory location of the argument to the stack location of the formal parameter as with call-by-value. However, any state changes that occur within the environment of the procedure are copied back into the memory location of the caller's arguments when the procedure returns. This is like call-by-reference since state changes are ultimately reflected back in the environment of the calling procedure.
- **Call-by-name.** This technique was used in Algol 60 and is not normally used anymore. The technique requires that local variables that have the same name as an actual argument be renamed to avoid naming conflicts.

Call-by-value and call-by-reference are the two most common calling conventions in use. C supports call-by-value only. Pascal and C++ support both, as does Ada. Ada also supports call-by-value-result (called *in-out parameters*)

Calling Conventions

Consider the following examples in C++ that illustrate the semantics of call-by-value and call-by-reference. The purpose of the swap function is to swap the values of two variables:

```
int x = 5, y = 10;
```

```
void swap(int a, int b) { // call by value
    int temp = a;
    a = b;
    b = temp;
}
```

What happens when you invoke the following

```
swap(x, y);
swap(4, 6);
```

In C++, you specify a reference variable using a special ‘reference operator’, denoted by ampersand ‘&’ following a type name. A reference type, such as `int&` means that an l-value is passed as the argument.

```
void swap (int& a, int& b) { // reference parameters are like ‘var’ parameters in Pascal
    int temp = a;
    a = b;
    b = temp;
}
```

The formal reference type parameters `a` and `b` are effectively *aliases* for the locations (l-values) of the actual parameters that are passed as arguments. What happens in the following cases?

```
swap(x, y);
swap(4, 6);
```

Note that the ‘&’ symbol in C++ is overloaded. When applied to the left side of a variable name, it means return the l-value of the variable. When applied to a type name in a formal parameter list, it means pass the argument by reference (l-value) not value (r-value).

Calling Conventions

In C, there are no reference types as there are in C++. However, C (and C++) have pointer types, and so you can get the effect of call-by-reference using pointers as arguments that are passed by value. That is, the r-value of a pointer is the address of some location. So, if you pass the value of a pointer, you get the l-value of some location, which is the same as passing the argument by reference (you pass its l-value in a pointer variable that is passed by value!)

```
void swap (int* a, int* b) {
    int temp = *a; // should check for a != NULL and b != NULL
    *a = *b;
    *b = temp;
}
```

Since the type of the formal parameters is “pointer to int” it is necessary to explicitly pass the address of a variable (the l-value) in a procedure invocation, using the “address-of operator”, also an ampersand ‘&’:

```
int x =5, y = 10;
swap(&x, &y); // pass addr(l-value) of x and y
```

What happens in the following case?

```
swap(4, 6);
```

Passing pointer arguments can be confusing and error prone at run-time. What happens if you inadvertently call the swap function, taking two pointer arguments, with one or both pointers equal to NULL? When you attempt to dereference the null-valued pointer, you get back the address 0, which is not a valid program address. There is no way to protect against this other than to write an if-statement to verify that the argument value(s) is not 0. But it could be non-null and still be an invalid address!

C/C++ programmers pass pointers because they are efficient (they fit into registers). However, they are the single biggest cause of bugs and errors in C/C++ programs: null pointers, dangling pointers, memory leaks, etc. Lots of hours are spent debugging such problems. The advantage of references in C++ is that the l-values for reference types are completely determined at compile-time. There is little chance that an l-value, which is known at compile time, will be NULL. Thus, reference types in C++ make programs safer—but C++ programmers still use pointers a lot!

Macro Expansion

An alternative to a procedure is a *macro*. A macro is a definition of a chunk of text that is to be substituted at the point where a macro call is made. Hence, macros define a textual substitution that does not obey the lexical scope rules of a typical procedure block, unless the macro is explicitly defined to introduce a new scope.

In C/C++, a macro is defined using a preprocessor directive. The preprocessor performs the textual substitution prior to the lexical analysis phase of the compiler

```
#define swap(a, b) temp = a; a = b; b = temp
```

Some key observations:

- although this looks similar to a function definition, there is no type information specified for each argument or for the result
- there is no scope defined
- the “body” of the macro is a list of statements, the last of which omits the ‘;’

A macro is used as follows:

```
int x = 5, y =10;
int temp;
...
swap (x, y); // textually expands to temp = x; x = y; y = temp;
```

Notice that the macro looks just like a procedure call, but in fact what happens is that the macro is “expanded” by substituting the variable names *x* and *y* for macro parameters *a* and *b*, then textually replacing the macro invocation with the statements defining the body of the macro.

An advantage of macros is that they are “untyped” and so you can use the *swap* macro to swap any pair of objects: integers, floating point values, etc., without having to write a function with explicit type parameters

Macro Expansion

Notice that when the swap macro is used, it is necessary to make sure that a local temp variable, of the appropriate type, has been declared in the scope in which the macro is invoked, prior to the use of the macro. In practice, this is not a very useful way to do things. There is also a problem with the use of the macro as part of a while, do, or if statement. Consider the following usage:

```
if (x < y)
    swap (x, y);
```

Which textually expands to:

```
if (x < y)
    temp = x;
    x = y;
    y = temp;
```

We would have to use `{}` around all invocations to the swap macro to avoid this type of subtle problem. Better yet, it is almost always advisable to define a macro procedure using a do-while statement:

```
#define swap(a, b) do { int temp = a; a = b; b = temp; } while (false) // leave off `;`
```

In this definition, a do-while statement that terminates after one iteration, is used to ensure that a single syntactic statement (and a new scope) is used to implement the body of the macro. So, the above if statement will now work properly, since do-while is a single syntactic statement, which does not need to be enclosed in `{}` itself, but allows a set of (compound) statements to occur in the body of the do-while statement.

The reason we use a do-while statement, is that we want to replace `swap(x,y)`, which looks like a single procedure call statement with another syntactic statement. That is a 1-for-1 substitution of a statement for a statement. The only way to do this is with a do-while statement. **Why would the following not work!?**

```
#define swap(a,b) { int temp = a; a = b; b = temp; }
```

Although it is desirable to define a new (nested) scope and declare the temp variable local to that scope, we have now committed `temp` to being of a specific type, namely, `int`. Our previously flexible swap macro that worked for any type, now only works for integer types, or types that can be type converted to an integer type (such as a `char`).

Overloaded Inline Functions

C++ improves on the macro idea in C (and other languages). Bjarne Stroustrup was also frustrated by the problems that macros introduce, so there are two features in C++ that make life simpler, safer, and more happy :-)

We can define *overloaded functions* in C++, which are procedures/functions that have the same name, but can take different argument types. We can also make these functions *inline*, by simply declaring them inline. For example:

```
inline void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
inline void swap(char& a, char& b) { char temp = a; a = b; b = temp; }
inline void swap(float& a, float& b) { float temp = a; a = b; b = temp; }
... // etc for any type you want.
```

An advantage of a macro, is that textual substitution does not incur the run-time cost of a procedure invocation. Since the above swap procedures have simple bodies consisting of three statements, we would like the compiler to *inline* a procedure call if possible. An inline procedure call is where the compiler (not the preprocessor) substitutes the code for the procedure at the point of invocation, preserving scope rules and type rules just as though a procedure call were actually made.

Notice that each swap function is identical, except for the type of the arguments and the type of the temp local variable. It turns out that C++ goes one step further towards making the life of a programmer simpler by defining *generic functions*, or *type template functions*. Templates allow us to parameterize a function definition with type information, and the compiler takes care of generating instances of the function that are overloaded, inline, and appropriately typed. For example:

```
template<class type> void swap(type& a, type& b) { type temp = a; a = b; b = temp; }
```

For each use of the swap function, the compiler will substitute the actual type of the arguments for the 'type' template parameter everywhere it occurs in the procedure definition. For example, swap(x,y) where x and y have type int becomes just what we want:

```
void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
```

Overloading Operator Functions in ML

By default, the usual arithmetic operators are defined as infix operator functions in ML.

For example, at the SML '`'` prompt, we can type in expressions in infix notation and get a result.

```
- 3 + 5;  
val it = 8 : int  
- 3.14 + 2.0;  
val it = 5.14 : real
```

Note from the result values, that ML deduces that it should use integer addition when the operands are integers, and floating point addition when the operands are reals. That is to say, the addition operator is **overloaded** to perform two different addition functions, depending on the (inferred) type of the operands. This is called *operator overloading* and is a form of *polymorphism*. We will examine other forms of polymorphism in programming languages throughout this course

Consider what happens when you mix operand types:

```
- 3.14 + 2;  
stdIn:1.1-2.4 Error: operator and operand don't agree [literal]  
  operator domain: real * real  
  operand:         real * int  
  in expression:  
    + : overloaded (3.14,(2 : int))
```

A *type error* is generated by the type system because there is a mismatch.

What are the types of the following ML expressions?

```
- fun square(x) = x * x;  
val it = fn : ??? -> ???
```

SML'97 treats this expression as type *fn: int -> int* by default. Earlier versions give an ambiguous type error.

User-Defined Infix Operators in ML

ML lets you define your own infix operator functions, using a builtin `infix` operator. We can then define the operator `xor` as a binary boolean function, implemented in terms of existing binary boolean operators.

```
- infix xor;
infix xor

- fun p xor q = (p orelse q) andalso not (p andalso q);
val xor = fn : bool * bool -> bool

- true xor false xor true;
val it = false : bool
```

Operator precedence in ML is specified in terms of a range of integer values: 0 - 9, with 0 the lowest precedence. When you declare an infix operator, the default precedence is zero. The `infix` directive causes *left associativity* by default. Operator `'+'` and `'-'` have precedence 6, `'*'` and `'/'` have precedence 7.

```
- infix 6 plus;
infix 6 plus
- fun a plus b = "(" ^ a ^ "+" ^ b ^ ";";
val plus = fn : string * string -> string

- infix 7 times;
infix 7 times
- fun a times b = "(" ^ a ^ "*" ^ b ^ ";";
val times = fn : string * string -> string

- "m" times "n" times "3" plus "i" plus "j" times "k";
val it = "(((m*n)*3)+i)+(j*k)" : string
```

Defining Infix Operators in ML

In addition to defining precedence level, we can also specify *right associativity* using the `infixr` directive.

```
- infixr 8 pow;
infixr 8 pow

- fun a pow b = "(" ^ a ^ "*" ^ b ^ ")";
val pow = fn : string * string -> string

- "m" times "i" pow "j" pow "2" times "n";
val it = "((m*(i**(j**2)))*n)" : string
```

It is also possible to revoke the infix status of an operator, using the *nonfix* directive. The function remains defined, however, it defaults to a binary *prefix* operator:

```
- nonfix pow;
nonfix pow

- "2" pow "4";
stdIn:48.1-48.12 Error: operator is not a function [tycon mismatch]
  operator: string
  in expression:
    "2" pow

- "m" times pow("2", pow("2", "10"));
val it = "(m*(2**(2**10)))" : string
```

Note by the way that all proper functions are really just prefix operators that operate on a n-ary list of arguments.

Compare with Operator Overloading in ANSI C++

C++, like ML but unlike C, allows you to overload operators for new types (classes) that are defined. For example, C++ does not by default define operators for complex numbers, but the programmer can define a new data type (class) `Complex`, and overload the standard arithmetic operators, giving them new meaning by defining their implementation in terms of the builtin operators for floating point arithmetic.

However, you cannot redefine their position, precedence, or associativity when used in expressions.

```
class Complex {
private:
    long double r; // real part
    long double i; // imaginary part
public:
    /* "Complex object constructor function" */
    Complex () { r = 0.0; i = 0.0; }
    Complex (double real, double imag) { r = real; i = imag; }
    ...
    /* "friend" functions can access the private data of a Complex object */
    friend Complex operator+ (Complex a, Complex b) { return Complex(a.r+b.r, a.i+b.i); }
    friend Complex operator- (Complex a, Complex b) { return Complex(a.r-b.r, a.i-b.i); }
    friend Complex operator* (Complex a, Complex b) { return ...; }
    friend Complex operator/ (Complex a, Complex b) { return ...; }
};
```

You can then declare “objects” of type `Complex`, and use it in expressions:

```
Complex x; // same as Complex x(0.0,0.0);
Complex a(1.0, 0.0);
Complex b(2.5, 3.0);
Complex c(2.0, 2.0);
...
Complex r = a + b * c; // a + (b * c) --- you can't change associativity in C++
```

Homework Assignment To Be Completed by Wed, 3 October

1. What is the difference in C++ between passing a parameter to a function using a pointer (e.g., `char*`) versus passing a parameter by reference (e.g., `char&`)?
2. Using C++, implement a swap procedure taking two pointer arguments pointing to null terminated character strings such that the two strings are swapped by passing pointers to simulate pass-by-reference. Print out the before and after values to demonstrate whether or not the swap was successful. For example:

```
char *a = "hello, world";
char *b = "goodbye, cruel world";
cout << "a = " << a << '\n';
cout << "b = " << b << '\n';
swap(a, b);
cout << "a = " << a << '\n';
cout << "b = " << b << '\n';
```

3. Explain what problem(s) arise if you try to implement a string swap function passing two “character references” (i.e., `char&`) arguments. Write a swap function in C++ that correctly swap two character strings?
4. Try implementing a swap function in Java that swaps two String objects. Explain why this is difficult to do in Java?
5. Explain the evaluation of the following expressions in terms of l-values and r-values. What is the final value of x?

```
int x = 0;
int *p = &x;
int **q = &p;
int ***r = &q;
**q = *p * 3;
*p++;
***r += *p + x * 2;
```