Overloaded Functions

Function overloading allows you to use the same name for different functions in the same program, provided each takes a different **type or number** of arguments. The pattern of arguments received by a function—which refers only to the number and types of the arguments and not the parameter names—is called its **signature**.

Here's an example. Both the `<cmath>` header and `<cstdlib>` declare `abs()` functions for returning the absolute value of a number. Here are the three functions in `<cstdlib>`:

```
int abs(int n);
long abs(long n);
long long abs(long long n);
```

Here are the four versions of the function in `<cmath>`:

```
double abs(double x);
float abs(float x);
long double abs(long double x);
double abs(T x); // a template called when no other matches
```

There are even versions for **complex numbers** in the header `<complex>`. The only difference between these functions is the **types of the parameters**. The compiler chooses **which version to call** by looking at the types of the arguments supplied.

- Called with an `int`, the compiler **calls** the version which **takes** an `int`.
- Called with a `double`, the compiler will choose the version from `<cmath>`, which takes a `double`.

*If you call `abs()` with an integer, and **only** include `<cmath>`, but forget `<cstdlib>`, then a special **generic version** of `abs()` that takes a type `T` parameter will be called. The difference between the generic version, and the overloaded `abs(int)` version, is that the generic version **always** returns a `double`, not an `int`.*

Overloading makes it easier for programmers to remember function names when the same operation is applied in slightly different contexts. C, which does not have overloading, requires different names for each different absolute value function: `iabs, fabs, dabs, labs, llabs, and so on.`

Overload Resolution

When you overload a function:

- The parameter **number** must differ, or
- The parameter **types** must differ, or
- The parameter type **order** must differ

You **cannot** merely change the return type of a function. That is an error.

To determine which function is called, your compiler follows a process called **overload resolution**. Resolving which version of the `abs()` function to call is easy, since it only takes one argument. Things are more complex when a function takes several arguments.

Here are the rules:

1. Functions **with the same name** are gathered into a **candidate set**.
2. The candidate set is narrowed to produce the **viable set**: those functions that have **the correct number of parameters** and whose parameters **could** accept the supplied arguments using standard conversions.
3. If there are any **exact matches** in the viable set, use that version.
4. If there are no exact matches, find the **best match** involving conversions. The rules for this can be quite complex. You can find all of the details in the C++ Primer, Section 6.6.

There are **two possible errors** that can occur **at the end** of the matching process:

- There are **no members** left in the viable set. This produces an **undeclared name** compiler error.
- The process **can't pick a winner** among several viable functions. This produces an **ambiguity** compiler error.

When this occurs, **the function definition is not in error**, but **the function call**.

Default Arguments

In your function declaration, you may indicate that certain arguments are optional by providing the parameter with a value to be used when no argument is passed in the call. These are called **default arguments**.

To indicate that an argument is optional, include an initial value **in the declaration** of that parameter in the function prototype. For example, you might define a procedure with the following prototype:

```
void formatInColumns(int nColumns = 2);
```

The `= 2` in the prototype declaration means that this **argument** may be omitted when calling the function. You can now call the function in two different ways:

```
formatInColumns();     // use 2 (default) for nColumns
formatInColumns(3);    // use 3 for nColumns
```

The `getline()` function which you have been using, actually has a third parameter, the terminating character, which is given the default value `'\n'` in its declaration.

Since most of the time you want to read an entire line, ending in a newline, that makes sense. However, if you supply a third argument, say `';'`, then `getline` will only read up to a `';'` instead of the entire line. This way you can use `getline` to read a series of delimited fields inside a single line.

Default Argument Rules

Here are the rules that determine the use of default arguments:

1. The default value appears **only in the function prototype**. If you repeat the default argument in the implementation file you will get a compiler error.
2. Parameters with defaults must **appear at the end of the parameter list** and cannot be followed by a parameter without a default. Here's a bad example:

   ```
   void badOrder(int a = 3, int b);    // how would you call this?
   ```

3. Default arguments are only used with value, **not reference** parameters. Here's another (bad) example:

   ```
   void badType(int& a = ????);    // what to set it to?
   ```

   Since a reference must refer to an `lvalue`, there is no way to specify which `lvalue` should be used when the function is called.

Prefer Overloading

Overloading is usually preferable to default arguments. Suppose for example, you wish to define a procedure named `setLocation()` which takes an `x` and a `y` coordinate as arguments.

You may write the prototype, **using default arguments**, like this:

```
void setLocation(double x = 0, double y = 0);
```

Now, the default location defaults to the origin `(0, 0)`. However, it **is possible** to call the function with **only one argument**, which is **confusing** to anyone reading the code. It is **better to just define a pair of overloaded** functions like this:

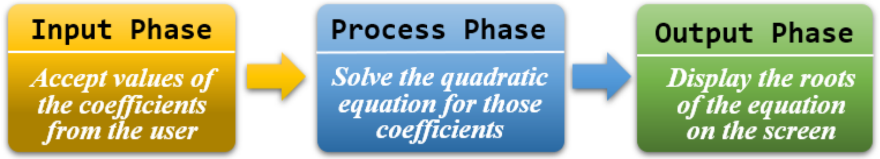
```
void setLocation(double x, double y);   // supply both
inline void setLocation() { setLocation(0, 0); }
```

The body of the second function, can just calls the first, passing `0`, `0` as the arguments. Since the function is very, very short, it can be marked `inline` which means it does not need to be defined inside the implementation file.

Data Flows

You may return more than one value from a function by using reference parameters to **return values through the argument list**.

As an example, suppose that you are writing a program to solve the quadratic equation below and you want to structure that program into three IPO phases like this:


Input Phase
Accept values of the coefficients from the user
Process Phase
Solve the quadratic equation for those coefficients
Output Phase
Display the roots of the equation on the screen

Using this plan, your `main` function might look like this:

```
1  int main()
2  {
3      double a, b, c, root1, root2;
4      getCoefficients(a, b, c);
5      solveQuadratic(a, b, c, root1, root2);
6      printRoots(root1, root2);
7  }
```

Here's what's happening:

- In line 4, the variables `a`, `b` and `c` are **output** parameters. Instead of passing values **into** the function, we're supplying three **variables** which will be filled up in the funciton itself.
- In line 5, `a`, `b` and `c` are **input** parameters. We are supplying their values to the `solveQuadratic` function so that it can do its work.
- The variables `root1` and `root2` are **output** parameters on Line 5; they will be filled in inside the function. On line 6, they are **input** parameters.

Output Parameters

Let's take a quick look at the code for `getCoefficients()` to see how output parameters work.

```
void getCoefficients(double& x, double& y, double& z)
{
    cout << "Enter 3 coefficients: ";
    cin >> x >> y >> z;
}
```

If a function **returns more than one** piece of information, then you can use **reference parameters** to return that information to the caller.

*Note that when you call `getCoefficients`, information **does not** flow from `main` **into** the function; instead, information **flows out of the function** back to `main`, through the three **output parameters** `x`, `y`, and `z`, which **are not new variables**, but are **new names** or **aliases** for the variables `a`, `b`, and `c` used when calling it.*

Instead of separate inputs, this function reads three variables using a **single input statement**. The values entered by the user must be **separated from each other by whitespace**, not commas. Spaces, tabs or newlines all work fine.

When **documenting your parameters**, you may **annotate** each of the parameters with the direction of the information flow: `@param[in]`, `@param[in,out]`, `@param[out]`. If you don't annotate the parameter, it is **assumed** to be an input parameter.

Input and Output Parameters

The `solveQuadratic()` function needs to use both input and output parameters. The arguments `a`, `b`, and `c` are **input** in this function, while `root1` and `root2` are **output parameters**, allowing the function to **pass back** the two roots to `main`.

```
void solveQuadratic(double a, double b, double c,     // input
                    double& r1, double& r2)           // output
{
    if (a == 0) die("a is 0; Illegal");   // no discriminate

    double discriminate = b * b - 4 * 1 * c;
    if (discriminate < 0) die("No real roots");

    double sqrtDisc = sqrt(discriminate);
    r1 = (-b + sqrtDisc) / (2 * a);
    r2 = (-b - sqrtDisc) / (2 * a);
}
```

Fatal Errors

Whenever this code encounters a condition that makes further progress impossible, it calls a function `die()` which prints a message and then terminates the program.

```
void die(const string& msg, int code = -1)
{
    cerr << "FATAL ERROR: " << msg << endl;
    exit(code);
}
```

- The `cerr` standard stream is similar to `cout`, but is reserved for reporting errors.
- The `exit()` function terminates a program immediately, using the value of the parameter to report the program status.
- The default error code is set to `-1`. If you want to use different error codes for different errors, just pass the code when you call `die()` (preferably as a `const`).

This function could be useful in many programs, so you might put it in a utility library.

Input-Output Parameters

We can use a single parameter for both input and for output. Consider `toUpperCase()` in Java. It takes a `String` as an argument, and returns a new, uppercase version of the original.

```
String str = "cat";
str = str.toUpperCase();  // CAT
```

This **builder method** does not (indeed, in Java, cannot) change its argument. However, that is a little inefficient, especially when assigned to the same variable, as we've done here.

Since C++ strings **may be modified**, we can write a more efficient version like this:

```
void toUpperCase(string& str)
{
    for (auto& c : str) { c = toupper(c); }
}
```

Here, `str` provides **both** input and output. We call this and **input-output** parameter. Because of that, it is passed by reference, **not** `const` reference. Note also that the loop variable `c` is a reference, not a value, so we can **modify the character it refers to**. Here's how to use it:

```
string str = "cat";
toUpperCase(str);  // Not str = toUpperCase(str)
```

Data Flow Checklists

Consider the `string::getline(in, str)` function:

- `in` is an **input-output** parameter. The function depends on the stream's initial state (formatting, etc.) and it is changed by calling the function (setting the error value).
- `str` is an **output only** parameter; it makes no difference what is inside `str` when you call the function—data **only flows out**.

The Java concept of data flow—parameters are input, return statements are output—is too simplistic for C++. In C++ (as in many other languages), parameters can be used as input, as output, or as a combination of both.

Use this checklist to determine the direction of data flow:

- ☐ Argument not modified by function: **input parameter**
- ☐ Argument modified, input value not used: **output** parameter
- ☐ Argument used and changed by function: **input-output** parameter

Use this checklist to determine **how to declare** the parameter variable:

- ☐ Output and Input-Output parameters: **by reference**
- ☐ Input primitive (built-in and enumerated) types: **by value**
- ☐ Input library and class types: by **const reference**

***Never** pass by value for class or library types.*

More Selection & Iteration

We have covered the basics of selection, iteration and functions in C++, but here are several additional features you might to use:

- The `switch` statement which provides an efficient multi-way branch based on the concept of an integer selector.
- The **conditional operator** which allows you to turn a 4-line *`if-else`* statement into a single, compact, expression.
- The *`do-while`*, (or **hasty**) loop, for when you want to leap before you look.

Let's take these in order.

The *switch* Statement

The *switch* statement implicitly compares an integral expression (called the selector) to a series of constants (called the **case labels**). Here's the syntax:

```
switch (selector)
{
    case constexpr1:
        statement;
        break;
    case constexpr2:
        statement;
        break;
    default:
        statement;
}
```

Here's how the `switch` statement works:

1. The `switch` **selector** is an **integral expression**.
2. It is evaluated and compared against the `case` label `constexpr1`, then `constexpr2`, and so forth. As indicated, each label **must be a constant** integer expression.
3. If a match is found for the selector, then **control jumps** to the first statement in the `case` block.
4. When control reaches the `break` at the end of the clause, it **jumps** to the statement that follows the entire `switch` statement.
5. The optional `default` specifies an action to be taken if **none of the constants** match the selector. If there is no `default` clause, the program simply continues on the line after the `switch`.

The constants in each `case` label statement must be **of integral type**. That means `char` and enumerated types are fine; **`string` or `double` are not**.

More on *switch*

Consider this code fragment inside a `switch`:

```
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    cout << "vowel";
    break;
```

As you can see, `break` statements are **not required** at the end of each `case`. If the `break` is missing, the program continues executing the **next clause** after it finishes the selected one. We say the `case` **falls-through**.

This is useful as shown here where the output is printed for all of the lower-case vowels.

```
case ' ': case '\t': case '\n':
    cout << "whitespace";
    break;
```

If there is nothing in the body of the case, it may be more readable to format it like the whitespace block shown here.

If there is **any code** inside a `case` block that falls through, most compilers will issue a warning. If you **intend** to fall through, and you want to suppress the warning, add a comment like this, just before the second case:

```
// fall through
```

A Few More Rules

- Two `case` labels may not have the same value
- A label must precede a statement or another `case` label. It may not be alone.
- Variables **may not** be defined inside one `case` block and then used in another.

The Conditional Operator

The conditional or selection operator uses two symbols: `?` and `:`, along with three different operands. It is also known as **the ternary operator** or **tertiary** operator for the number of operands. The general form is

```
(condition) ? true-result : false-result
```

The parentheses are not technically required, but programmers often include them to make things clearer. Here's how the conditional operator works:

1. The condition is evaluated.
2. If the condition is `true`, ***`true-result`*** is evaluated and used as the expression's value.
3. If the condition is `false`, then ***`false-result`*** is used as the expression's.

Here are two examples:

```
1  int largest = (x > y) ? x : y;
2  cout << ((cats != 1) ? "cats" : "cat") << endl;
```

- **Line 1** assigns the larger of `x` or `y` to the variable `largest`, without the need for a multi-line *`if`* statement.
- **Line 2** prints `"cat"` if there is only one cat, and `"cats"` otherwise.

Note that when you use the conditional operator as part of an output statement, you **must** parenthesize the whole expression, since it has very low precedence.

A Hasty Loop

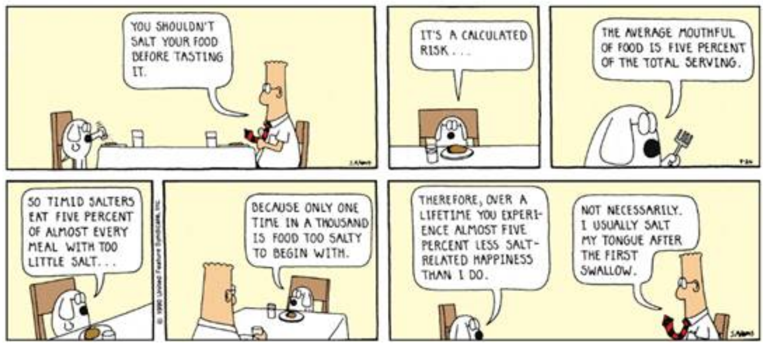
In addition to *for* and *while*, C++ has a loop that tests its condition **after** the loop body completes. The do-while loop always executes the statements inside its body at least once.

```
do
{
    // statements
}
while (condition);
```

The body of the ***do-while*** loop appears between the keywords **do** (which precedes the loop body) and **while**. The body of the ***do-while*** loop can be a single statement, ending with a semicolon, or it can be a compound statement enclosed in braces.

In the ***do-while*** loop, the condition is **followed by a semicolon**, unlike the ***while*** loop, where following the condition with a semicolon leads to subtle, hard tofind bugs.

The ***do-while*** loop is often employed by beginning programmers because it seems more natural. If you find yourself in this situation, think twice. 99% of the time, a ***while*** loop or a ***for*** loop is a better choice than a ***do-while***. In fact, except for salting your food...


YOU SHOULDN'T SALT YOUR FOOD BEFORE TASTING IT.
IT'S A CALCULATED RISK...
THE AVERAGE MOUTHFUL OF FOOD IS FIVE PERCENT OF THE TOTAL SERVING.
SO TIMID SALTERS EAT FIVE PERCENT OF ALMOST EVERY MEAL WITH TOO LITTLE SALT...
BECAUSE ONLY ONE TIME IN A THOUSAND IS FOOD TOO SALTY TO BEGIN WITH.
THEREFORE, OVER A LIFETIME YOU EXPERIENCE ALMOST FIVE PERCENT LESS SALT-RELATED HAPPINESS THAN I DO.
NOT NECESSARILY. I USUALLY SALT MY TONGUE AFTER THE FIRST SWALLOW.

which should **always be done before tasting**, there are relatively few other situations where a test-at-the-bottom strategy is superior to "looking before you leap."

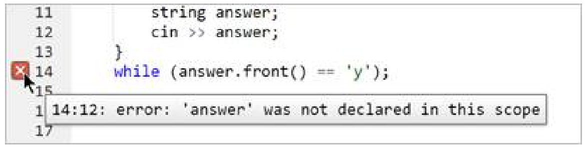
Confirmation Loops

When you make a withdrawal at your ATM, before your card is returned, the machine will ask you "Do you want to make another transaction?" This is a **confirmation loop**, and the ***do-while*** loop seems ideal for solving this problem.

However, there are still some things you need to watch out for. Considerthis code:

```
do
{
    completeSomeTransaction();
    cout << "Do you want another transaction? ";
    string answer;
    cin >> answer;
}
while (answer.front() == 'y');
```

While this **looks reasonable** (other than not providing for the empty string or an upper-case 'Y'), it actually **won't compile**. When you get to the **loop condition**, the `string` variable `answer` has **gone out of scope**.



```
        string answer;
        cin >> answer;
    }
    while (answer.front() == 'y');
```

```
14:12: error: 'answer' was not declared in this scope
```

To fix this, you have to move the initial declaration for `answer` before the `do` statement, which is not quite as clear. So, even in this natural use-case, the ***while*** loop is a little more efficient.