ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo

Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Introductory Project.
Assistances and comments will be acknowledged.

Contents

0 Laboratory introduction

In this laboratory we will cover the following topics:

1. laboratory safety,
2. project requirements,
3. …

0.1 Laboratory safety

The Helix Laboratory in RCH 108 is in the basement floor of the Ron Coutts Hall. The computer laboratory has two exits and there are four exits from the building, as shown in Figure 1. There are emergency information posters at each of the laboratory entrances and the location of the first-aid kit is listed on the *First Aid Emergency Procedures* poster. In the small adjacent room at the back of the laboratory is a telephone that can be used in an emergency. There are four fire extinguishers located as you move towards the four building exits.

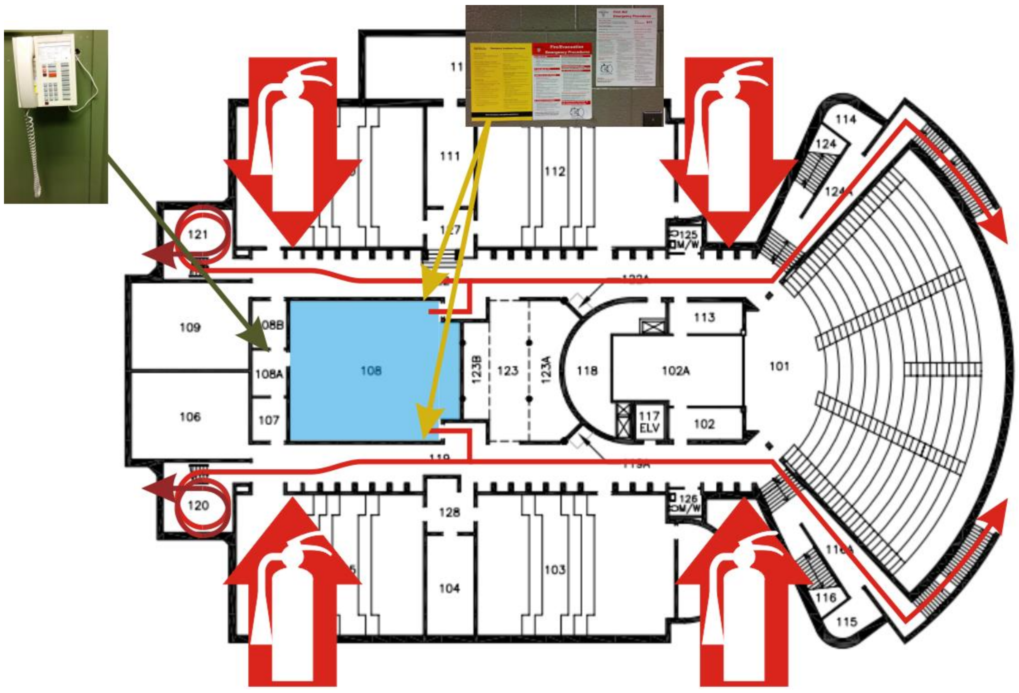


Figure 1. Room and building exits.

0.2 Project requirements

The projects for ECE 250 are meant to be completed by each student individually. You may work together, but each student should be the sole author of the code he or she submits. There are six laboratories numbered Laboratory 0 through Laboratory 5 and there is an introductory project followed by five projects submitted for grading. The laboratories are scheduled during the even weeks of the term. Laboratory 0 introduces you to C++ and provides you with an introductory project, while Laboratories 1 through 5 are associated with Projects 1 through 5, respectively. You will be able to seek help from the laboratory instructor and teaching assistants during that time. Projects 1 through 4 are due at 10:00 PM of the Tuesday immediately following the laboratory corresponding to that project. Project 5 is due at midnight on the last day of class. All submissions must be through the drop boxes on Learn. The drop boxes are left open for late submissions until 6:00 AM the next morning.

While you should be able any integrated development environment (IDE) you wish to develop solutions to these projects, you must be aware that all submissions will be graded on the Linux computer `ecelinux.uwaterloo.ca`. Your code must run execute on this machine for it to be graded.

Further details about the projects may be found at http://ece.uwaterloo.ca/~dwharder/aads/Projects/.

0.3 Pass-by-value and pass-by-reference

As a quick review, by default in C and C++, arguments are passed by value. For primitive data types, the compiler allocates new memory for the parameters and copies the values of the arguments to those parameters. On the other hand, if you have a class, a pass-by-value will allocate memory for a new instance of that object and just copy over the values of the member variables. If you change a parameter that has been passed by value, the argument is unaffected. In this example, the origin is unchanged after the call to `move(…)`.

```
class coord {
    public:
        double x;
        double y;
};

void move( coord pt, double dx, double dy ) {
    pt.x += dx;
    pt.x += dy;
}

int main() {
    coord origin;
    origin.x = origin.y = 0.0;

    move( origin, 3.2, 4.5 );

    return 0;
}
```

A pass-by-reference in C++ indicates that any reference to the parameter is a reference to the original argument. Thus, if we were to re-implement the move function as

```
void move( coord &pt, double dx, double dy ) {
    pt.x += dx;
    pt.x += dy;
}
```

then the origin would be shifted to (3.2, 4.5) after the call to the function `move(…)`. Another example where pass-by-reference is necessary is a swap function. The function

```
void swap( int x, int y ) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

will only copy the values of the arguments to the parameters x and y and the swap will leave the original values unchanged.

If, however, the arguments were passed by reference, as in

```
// Swap the two arguments
void swap( int &x, int &y ) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

then these would swap the values of the original arguments.

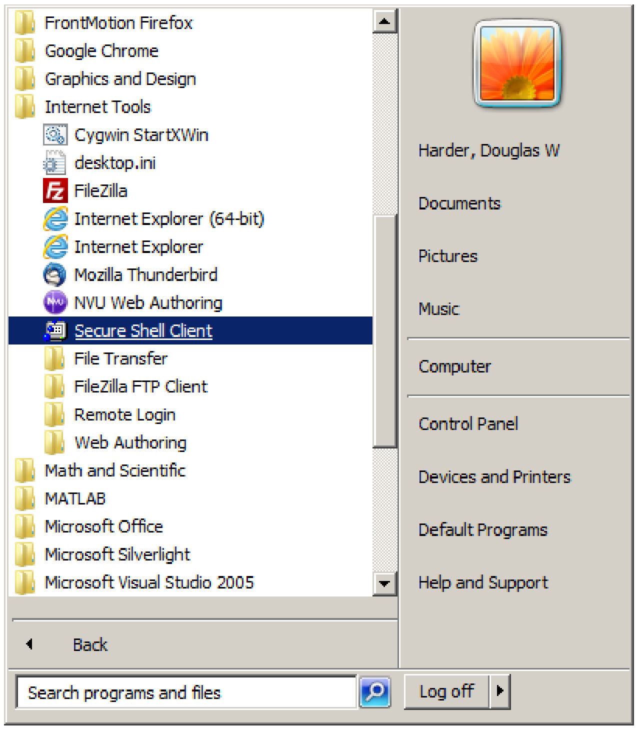
One benefit of pass-by-value is that the function cannot change the argument that is passed, while a pass-by-reference allows the function to modify the argument. One cost of pass-by-value for objects is that a copy of the object must be made—something that may be very expensive, and therefore pass-by-references is generally desirable for passing objects; however, now every function call can potentially change the function. To avoid this, a function can use pass-by-constant-reference. Thus, if a function is

```
// Calculate the distance between the two coordinates
double distance( coord const &p1, coord const &p2 ) {
    return std::sqrt( (p1.x - p2.x)*(p1.x - p2.x)
                    + (p1.y - p2.y)*(p1.y - p2.y) );
}
```

Now, if a programmer accidently introduced a change to the distance function that modified one of member variables of either argument, the compiler would return an error.

0.4 Remote login and Unix

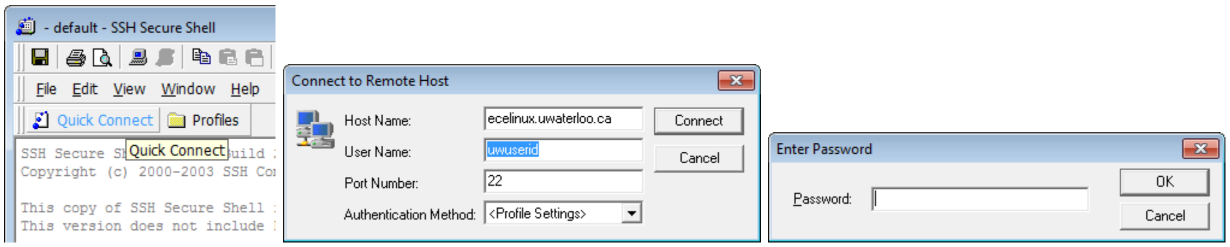
On any engineering computer, you can find the **Secure Shell Client**, often under the **Internet Tools** folder in the start menu; although, it may be also accessible under the **Remote Login** folder.



This will open a window and you can select Quick Connect in the menu. This opens a dialog **Connect to Remote Host** where you will enter

1. the **Host Name**, in this case `ecelinux.uwaterloo.ca`, and
2. your **User Name**, in this case your uWaterloo user ID, for example, `dwharder`.

After selecting **Connect**, this will open a second dialog labeled **Enter Password**; use your Nexus password.

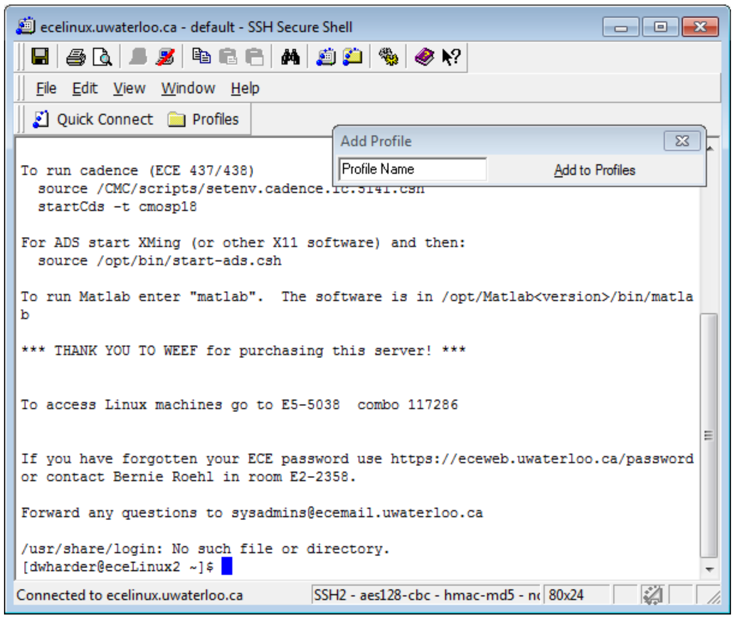


```
SSH Secure Sh     uild
Copyright (c) 2000-2003 SSH Co

This copy of SSH Secure Shell
This version does not include
```

Host Name: ecelinux.uwaterloo.ca
User Name: uwuserid
Port Number: 22
Authentication Method: <Profile Settings>
Password:

You have now established a remote shell user interface. The URL `ecelinux.uwaterloo.ca` is an alias for approximately ten different computers, named `ecelinux1`, `ecelinux2`, etc., and you will be remotely logged onto one of these chosen at random (based on load). The Secure Shell Client will give you the option of saving your Host Name and User Name under a profile that you can then select from the available profiles, as opposed to always entering this information each time you want to log in.


ecelinux.uwaterloo.ca - default - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Add Profile
Profile Name
Add to Profiles
Connected to ecelinux.uwaterloo.ca
SSH2 - aes128-cbc - hmac-md5 - n
80x24

```
To run cadence (ECE 437/438)
  source /CMC/scripts/setenv.cadence.IC.5141.csh
  startCds -t cmosp18

For ADS start XMing (or other X11 software) and then:
  source /opt/bin/start-ads.csh

To run Matlab enter "matlab".  The software is in /opt/Matlab<version>/bin/matla
b

*** THANK YOU TO WEEF for purchasing this server! ***


To access Linux machines go to E5-5038  combo 117286


If you have forgotten your ECE password use https://eceweb.uwaterloo.ca/password
or contact Bernie Roehl in room E2-2358.

Forward any questions to sysadmins@ecemail.uwaterloo.ca

/usr/share/login: No such file or directory.
[dwharder@eceLinux2 ~]$ 
```

Now that you are in Unix, you session is associated with a *working directory*. If you *print* the working directory, you will see something like

```
$ pwd
/home/dwharder
```

This is your *home* directory—each user will have their own directory under `/home`. If you want to *list* the files and subdirectories of the working directory, use the `ls` command:

```
$ ls
files and directories are listed here...
$
```

If we want to make a new directory, we use the `mkdir` command, and to *change* the working *directory* to a sub-directory, use the `cd` command:

```
$ mkdir ece250
$ ls
ece250 and other files and directories...
$ cd ece250
$ ls
$ mkdir lab0
$ ls
lab0
$ cd lab0
$ pwd
/home/dwharder/ece250/lab0
```

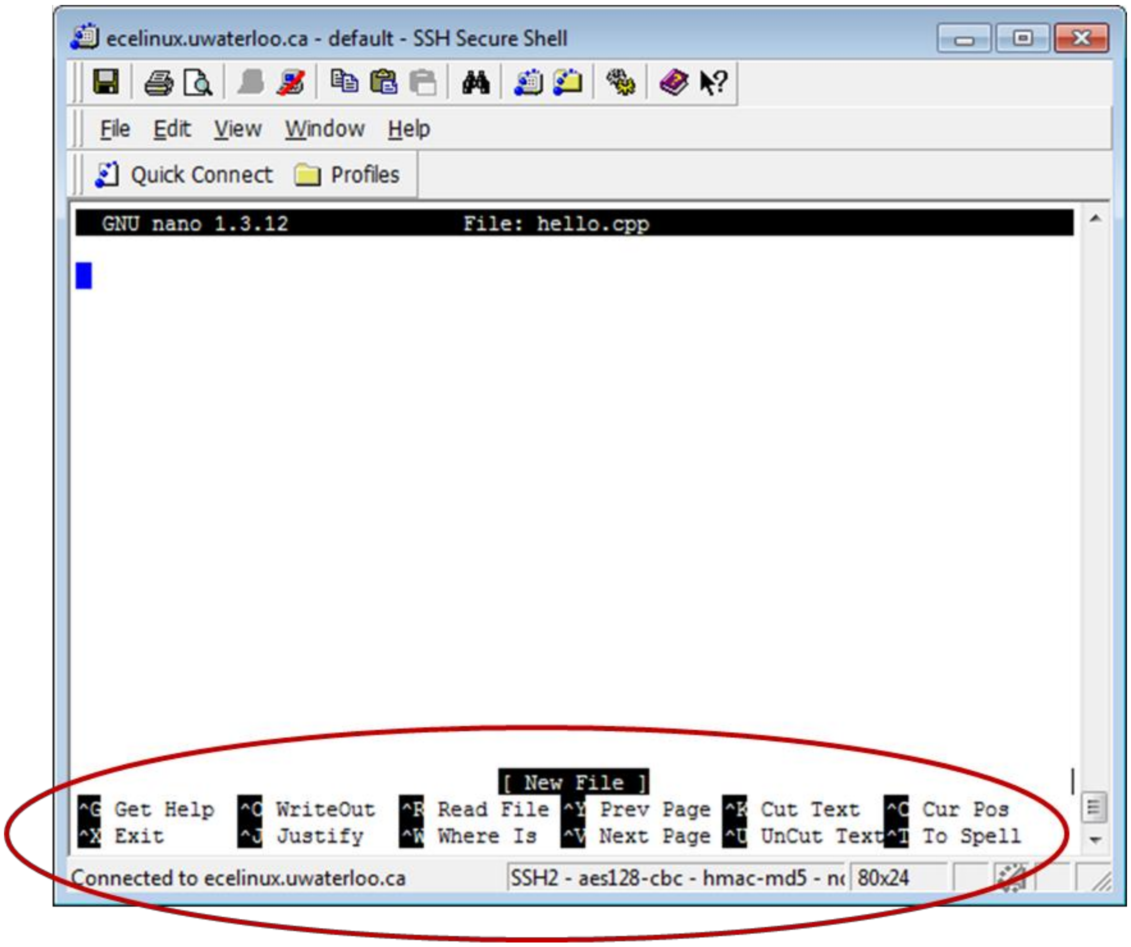
Important: If you type the command cd with no arguments, this returns you to your home directory.

0.5 Hello world!

We will now implement the standard *Hello world!* program in C++ using Unix. At the command prompt, we need to edit a file. Let's call it `hello.cpp`:

```
$ nano hello.cpp
```

This opens a very simple text editor in Unix. All the commands are control characters and they are listed at the bottom of the editor window:


ecelinux.uwaterloo.ca - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
Connected to ecelinux.uwaterloo.ca
SSH2 - aes128-cbc - hmac-md5 - n
80x24

```
  GNU nano 1.3.12               File: hello.cpp

                              [ New File ]
^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

The commands are listed here for convenience:

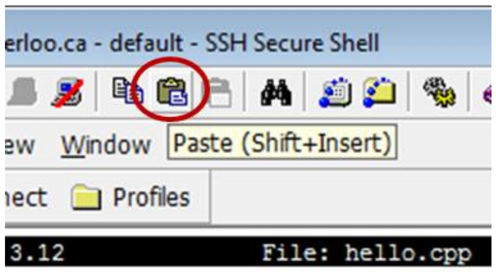
```
Ctrl-g Get help
Ctrl-o Write out
Ctrl-r Read a file
Ctrl-x Exit
Ctrl-y Previous page
Ctrl-v Next page
Ctrl-k Cut text
Ctrl-u Paste text
Ctrl-c Cursor position
Ctrl-j Justify
Ctrl-w Search
Ctrl-t Spell checker
```

Now, we must copy the code and paste it into the editor. Take the code

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

and copy it, past it into the terminal. To do this, you must use Ctrl-c in Windows and then use *Edit→Paste* in the SSh client.


erloo.ca - default - SSH Secure Shell
ew Window Paste (Shift+Insert)
nect Profiles
3.12 File: hello.cpp

This inserts your code as if you typed yourself. Next, save the file (Ctrl-o) and exit (Ctrl-x).

If we now list the contents of the current working directory, we will see the file:

```
$ ls
hello.cpp
```

We now want to use g++ to compile the source code into an *executable file* or *program*. To do this, we use the g++ command, but while we do this, we will learn about command completion. Type:

```
$ g++ h
```

and now hit the Tab key. Because there is only one file starting with "h" in the current directory, the shell will complete this command to

```
$ g++ hello.cpp
```

If you execute this command, it creates a program named `a.out`:

```
$ ls
a.out    hello.cpp
```

If you want a different name, you can use the `-o` option for `g++`, which allows you to specify an output file. For example, you could use

```
$ g++ -o hello hello.cpp
```

and now the executable would be the file `hello`.

In order to execute the file, we must use the invocation

```
$ ./a.out
Hello world!
$
```

Normally, you might think that you only have to give the name of the executable; however, Unix will only execute commands located in specific directories, and this does not normally include the current working directory. Therefore, you must say "in the current working directory (`.`), execute the file `a.out`".

In the Unix command prompt, . is an alias for the current working directory, and .. is an alias for the parent directory. If you type

```
$ cd .
```

your current working directory will be unchanged. If you type

```
$ cd ..
```

your current working directory will be changed to the parent of the current working directory.

0.6 Creating a project

We will now discuss creating a new project in both Unix and in Microsoft Visual Studio. If you are using a different IDE or platform, many of the actions will be similar to these described here.

0.6.1 Unix

In Unix, you will simply edit files and compile therm. For this project, the source file will be one called `Array.h`. You can edit this file with nano:

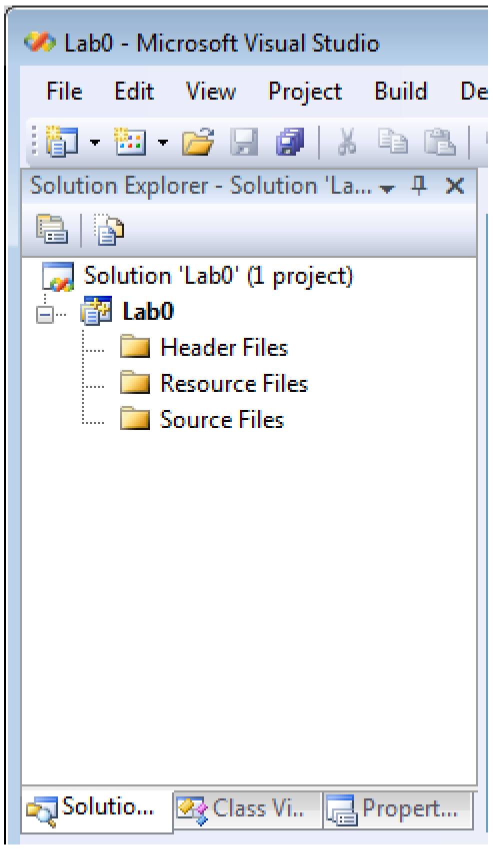
```
$ nano Array.h
```

Once we have created this file, we will need to create an executable: a file that uses this array class and that contains a function with the signature `int main()`.

```
$ nano main.cpp
```

0.6.2 Microsoft Visual Studio

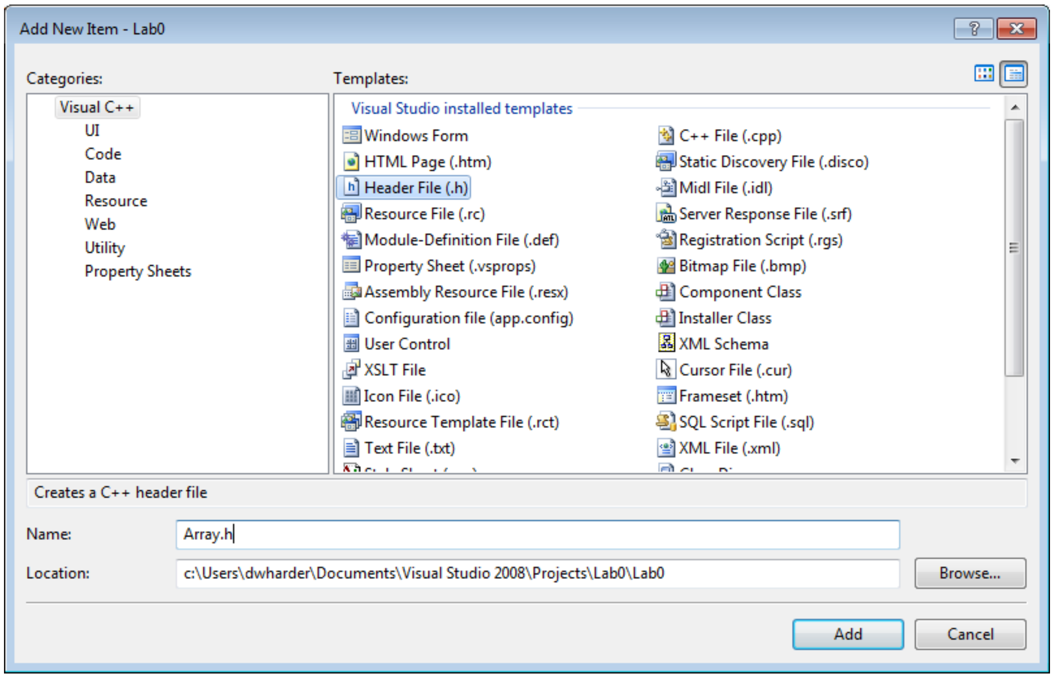
We will now look at how we can create a project in Microsoft Visual Studio, but most IDEs will have equivalent actions. First, we must launch Visual Studio by selecting it from the Start menu. Create a new project and select the project type to be ***General***. Use the ***Empty Project*** installed template, and use **Lab0** as the project name. This will create a project that has locations for header, resource and source files. We will use the first and third.



We will create a header file by:

1. Right-clicking on the ***Header* Files** folder,
2. Selecting *Add →New Item…* which brings up the ***Add New Item*** dialog, and
3. Select the ***Header file (.h)*** template, enter the name `Array.h,` and select ***Add***.

The window is shown here:



Creates a C++ header file

Name: Array.h

Location: c:\Users\dwharder\Documents\Visual Studio 2008\Projects\Lab0\Lab0

We can now start editing the file. Later, we will have to create a source file that we can compile into an executable. To do this, we would

We will create a header file by:

1. Right-clicking on the ***Header* Files** folder,
2. Selecting *Add →New Item…* which brings up the ***Add New Item*** dialog, and
3. Select the ***C++ file (.cpp)*** template, enter the name `main.cpp,` and select ***Add***.

0.63 Summary of creating a project

We have described how to edit files and create projects in Unix and Visual Studio. In any integrated development environment (IDE), most of the work is done for you; whereas, in Unix, you will be responsible for many more tasks that are taken care of by an IDE.

0.7 Creating our `Array.h` header file

We will start with a minimal array class and build from it, adding features both to what we have already implemented and including new features throughout subsequent sections. We will start with a simple class that allocates memory for an array class, allows the user to append entries into the array, and allows the user to return the number of entries in array (the *size*).

```
#ifndef ARRAY_H
#define ARRAY_H

// Array class definition
class Array {
    private:
        int array_capacity;
        int *internal_array;
        int array_size;
    public:
        // Member function declarations
        Array( int );
        int size() const;
        void append( int );
};

// Member function definitions

// The constructor
//  - create an instance of the array by
//     1. initializing the array capacity to 'n',
//     2. allocating memory for an array of the given capacity, and
//     3. initializing the array size to 0

Array::Array( int n ):
array_capacity( n ),
internal_array( new int[array_capacity] ),
array_size( 0 ) {
    // does nothing
}

// Returns the number of entries in the array
int Array::size() const {
    return array_size;
}

// Append a new entry into the array
//
// Note that
//  - when the array size is k, the next item should be appended in location k
//    and the size is then k + 1
void Array::append( int obj ) {
    // currently, entries 0, ..., array_size - 1 are occupied
    internal_array[array_size] = obj;
    ++array_size;
}

#endif
```

0.8 Improving our implementation

We will consider three changes to our implementation above:

1. adding a default value for the array capacity,
2. ensuring that the array capacity is positive, and
3. preventing the user from appending beyond the capacity of the array.

0.8.1 Default values

In our current design, the user must provide a value for the capacity of the array; for example,

```
        Array info( 10 );
```

If a user was to declare an instance of our array class without a value, say

```
        Array info2;
```

This would cause a compilation error: there is no constructor that takes no arguments (or, in other words, there is no *default constructor*).

Recall that a default constructor is provide only if no other constructors are explicitly provided where the default constructor simply initializes all of the member variables to their default values (for primitive data types) or calls their default constructors (for classes))

If you attempt to call a function for which there is no declaration, you will get an error. Using `g++`, you will get an error message such as

```
main.cpp: In function 'int main()':
main.cpp:5:  error: no matching function for call to 'Array::Array()'
Array.h:116: note: candidates are: Array::Array(int)
Array.h:73:  note:                 Array::Array(const Array&)
```

Note that it gives you both the file name and the line on which the error occurs. The second constructor is the default *copy constructor* (about which we will see more later). In Microsoft Visual Studio, the error message is

```
1>c:\users\dwharder\documents\visual studio 2008\projects\lab0\lab0\main.cpp(8) :
     error C2512: 'Array' : no appropriate default constructor available
```

Again, it gives you the file and line number (8) together with an error message. There are two options the designer has to deal with this issue:

1. Consider this a user error.
2. Provide a default value for the parameter.

For this example, we will look at how to add a default value. In the class definition, we can provide a default value in the member function declaration by using the assignment operator.

```
class Array {
    private:
        int array_capacity;
        int *internal_array;
        int array_size;
    public:
        Array( int = 10 );
        int size() const;
        void append( int );
};
```

This must be given in the class definition—you should not try to repeat the default value in the member function definition—it remains unchanged:

```
Array::Array( int n ):
array_capacity( std::max( 1, n ) ),
internal_array( new int[array_capacity] ),
array_size( 0 ) {
    // does nothing
}
```

0.8.2 Correcting problems in the constructor

In our class, we declared the type of the argument to be `int`. This, unfortunately, allows the user to specify a negative argument. There are three possible solutions to this:

1. Change the type from a signed integer to an unsigned integer.
2. Throw an exception from within the constructor.
3. Use a different value.

In the first case, we could use a type such as `unsigned int`, or even better `size_t`. Unfortunately, this does not prevent the user from declaring an array of size `0`.

Aside: The type size_t is a type that changes based on the characteristics of the platform. It is the datatype capable of storing the maximum index of the theoretically largest possible array on the platform. This would be 8 bytes (64 bits) on a 64-bit machine and 4 bytes in size on a 32-bit machine.

It is also possible to throw an exception from the constructor (and we will discuss this later); however, the destructor is not called, and any allocated memory or resources will have to be deallocated before the exception is thrown.

In the third case, we could just correct the issue by using a default size (say 1) if the user specifies an invalid array size. In our case, we must correct this before the memory for the array is allocated. The following implementation fails, as `new` is still called with the zero or negative value.

```
Array::Array( int n ):
array_capacity( n ),
internal_array( new int[array_capacity] ),
array_size( 0 ) {
    if ( array_capacity <= 0 ) {
        array_capacity = 1;
    }
}
```

Instead, we must fix the problem as soon as we initialize the capacity:

```
Array::Array( int n ):
array_capacity( std::max( 1, n ) ),
internal_array( new int[array_capacity] ),
array_size( 0 ) {
    // does nothing
}
```

The standard library max function is declared in the algorithm library, and therefore we must include

```
#include <algorithm>
```

at the top of the file `Array.h`.

0.8.3 Preventing the user from appending beyond the capacity of the array

Our append function simply appends the new value to the end of the array; however, no attempt is made to check whether or not we are still within the bounds of the allocated memory. Thus, a function such as

```
int main() {
    Array info( 10 );

    for ( int i = 0; i < 20; ++i ) {
        info.append( i );
    }

    std::cout << "The size is " << info.size() << std::endl;
}
```

would happily write to entries such as `internal_internal[10]` and beyond. However, when the initialization

```
internal_array( new int[array_capacity] ),
```

was executed in the constructor, the operating system found an available block of 40 bytes of memory and returned the address of the first byte. Thus, suppose the available memory was found at memory location `0x0039af78`, as shown in Figure 2.

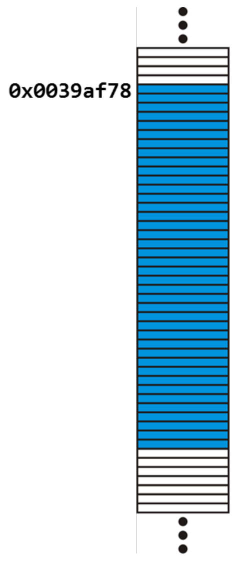


Figure 2. 40 bytes allocated at address `0x0039af78`.

If you access `internal_array[n]`, the compiler simply accesses the integer at location

`internal_array + 4 x n`

and in this example, that would be

`0x0039af78 + 4 × n`

Thus, an access to array entry 5 would access memory location `0x0039af8c`, and accessing array entry 10 would access the memory location `0x0039afa0`; however, the second address is beyond the memory of memory allocated.

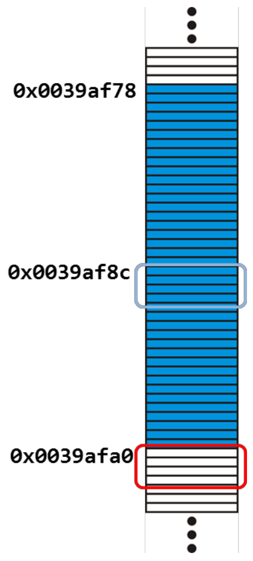


Figure 3. Accessing entries 5 and 10.

Two things may happen:

1. That memory may not be assigned to your executing program, so your executing program will be terminated.
2. The memory is assigned to your executing program, in which case
    a. you may not be using it, so no problem, but
    b. you may be using that memory to store other data, in which case, that data will be overwritten.

Thus, there are many things we could do if the array is full, including to

1. ignore the append,
2. not append and signal the user that the array is full, or
3. throw an exception.

Aside: If two separate executing programs were accessing the same array in the same memory location, it might be possible to put one executing program to *sleep* until the other executing program makes a location in the array available. You will learn about this in your operating system course.

In this example, we will simply signal the user that it was not possible to append the value. To do this, we must change the member function declaration in the class definition to indicate that it is now returning a `bool` as well as the member function definition:

```
class Array {
    // ...
        bool append( int );
    // ...
};

bool Array::append( int obj ) {
    if ( array_size == array_capacity ) {
        return false;
    }

    // currently, entries 0, ..., array_size - 1 are occupied
    internal_array[array_size] = obj;
    ++array_size;

    return true;
}
```

0.8.4 Summary of updated functionality

In this section, we have updated the behavior of the constructor and the append member function. We will continue by adding three more member functions.

0.9 Adding additional functionality

Currently, we have one member function that returns the number of entries inserted into the array, but the user may want additional information:

1. What is the capacity of the array?
2. Is the array empty?
3. Is the array full?

All three functions will be read-only (declared `const`), and the first will return an `int`, while the latter two will return a `bool`. The member function declarations must be added to the class definition, and member function definitions must be included below.

```
class Array {
    private:
        int array_capacity;
        int *internal_array;
        int array_size;
    public:
        Array( int = 10 );

        // Accessors
        int size() const;
        int capacity() const;
        bool empty() const;
        bool full() const;

        // Mutators
        void append( int );
        void clear();
};
```

While the implementation of `int capacity() const` is essentially identical to the implementation of `int size() const`, there are numerous ways we could implement the Boolean-valued functions. The most obvious may be something like

```
bool Array::empty() const {
    if ( array_size == 0 ) {
        return true;
    } else {
        return false;
    }
}
```

However, consider the following statement:

```
int a = 4, b = 7;
bool n = a == b;
```

We note that `==` is an operator like `+`: it compares the two operands and returns `1` if they are equal, and `0` if they are different. The values `true` and `false` are simply aliases for the bytes `0x00` and `0x01`, and in C++, for a conditional statement, any non-zero value is interpreted as *true* and any zero is interpreted as *false*. Thus, the statement `array_size == 0` simply evaluates to either `true` or `false`, so why not just return that value?

```
    bool Array::empty() const {
        return (array_size == 0);
    }
```

Note: the parentheses around the equality statement are not necessary, but they simply make it easier for the reader to note the operation. This is clearer than either

```
    return array_size == 0;
    return array_size==0;
```

As an example of how conditional statements simply evaluate the expression, consider

```
    int i = 10;

    while ( i ) {
        --i;
        array[i] = 0;
    }
```

This will initialize the entries of the array from 9 down to 0 to `0`. This is not at all supposed to suggest what one may consider a "good" programming technique, it is merely given to demonstrate the evaluation of conditional statements in C and C++.

Once we have implemented the bool full() const function, we really should go back and update the append member function:

```
    bool Array::append( int obj ) {
        if ( full() ) {
            return false;
        }

        // currently, entries 0, ..., array_size - 1 are occupied
        internal_array[array_size] = obj;
        ++array_size;

        return true;
    }
```

It is much easier to read the intention of the programmer by using the member function full. Another programmer reading the statement `array_size == array_capacity` must first determine the purpose of each member variable and then determine the significance of their values being equal.

Next we will use the `void clear()` function to discuss some C-specific features of arrays and to demonstrate how to read errors messages from the compiler. First, to clear the array, it is enough to reset the array size variable to zero. Then, with the next append, the new item will overwrite any item that had previously been written there.

Now, suppose that the member function declaration in the class definition did not declare `clear()` to be read only (`const`) but the member function definition did. In this case, the compiler will point out that it cannot find a declared member function with the name of the member function in the definition. In Visual Studio, the error message is

```
'void Array::clear(void) const' : overloaded member function not found in 'Array'
```

In `g++`, the error message is

```
Array.h:36: error: prototype for'void Array::clear() const'
does not match any in class 'Array'
Array.h:17: error: candidate is: void Array::clear()
```

Because the `void clear()` member function assigns the array size member variable, if both are declared const, the compiler would now issue an error that a member variable is being assigned in a read-only (`const`) member function. In Visual Studio, the error message is:

```
l-value specifies const object
```

In `g++`, the error message is

```
Array.h:37: error: assignment of data-member
'Array::array_size' in read-only structure
```

Just like you cannot assign to any member functions in a read-only member function, the only member functions you can call from a read-only member function are other read-only member functions. For example, if you were to call `void clear()` inside the `bool empty() const` function, the error message you would get is

```
Array.h: In member function 'bool Array::empty() const':
Array.h:178: error: passing 'const Array' as 'this' argument of
'void Array::clear()' discards qualifiers
```

0.10 Four statistical functions

We will now look at three statistical functions that describe the contents of the arrays:

1. calculating the sum of the entries,
2. calculating the average (or *mean*),
3. calculating the sample variance, and
4. calculating the sample standard deviation.

Even though the array stores integers, the average, variance and standard deviation of these integers will most likely be real numbers (the average of 1 and 2 is 1.5). Consequently, these functions must return double-precision floating-point numbers:

```
class Array {
    private:
        int array_capacity;
        int *internal_array;
        int array_size;
    public:
        Array( int = 10 );

        // Accessors
        int size() const;
        int capacity() const;
        bool empty() const;
        bool full() const;

        int sum() const;
        double average() const;
        double variance() const;
        double std_dev() const;

        // Mutators
        void append( int );
        void clear();
};
```

Note: we should also be careful about having sum return an integer, as summing integers may result in an overflow. For example, if there are two entries 2147483647 and 1, the calculated sum would be the negative integer –2147483648. Consequently, in other applications, it may be necessary to check to ensure that arithmetic overflow or underflow does not occur. In this case, we will be content to return an integer and ignore this issue.

Given an array $a = (a_1, \ldots, a_N)$ of N values, we define

1. the sum as $\sum_{k=1}^{N} a_k$,
2. the average as $\bar{a} = \frac{1}{N}\sum_{k=1}^{N} a_k$,
3. the sample variance as $s_a^2 = \frac{1}{N-1}\sum_{k=1}^{N}\left(a_k - \bar{a}\right)^2$, and
4. the sample standard deviation as $s_a = \sqrt{s_a^2}$.

The sum of an empty array should be 0. At the very least, the sum function will require a repetition statement (`for` loop) that iterates through the entries. In this case, the ideal form of such a loop is

```
for ( int i = 0; i < size(); ++i ) {
    // do something
}
```

In each case, these functions should use functions we have already defined:

1. `average()` should call `sum()` (we will assume that sum does not over- or underflow),
2. `variance()` should call `average()`, and
3. `std_dev()` should call `variance()`.

It is always best to re-use a function you have already implemented as opposed to recreating the behavior you have already defined elsewhere.

You will note that the average is not defined if $N = 0$, and neither of the sample variance or standard deviation are defined if $N \le 1$. We cannot return a default value and therefore we must signal to the user that an exception to the expected behavior has occurred. To achieve this, examine the `Exception.h` file in the source directory of this project. The classes of interest include:

```
class exception {
    // empty class
};

class underflow : public exception {
    // empty class
};
```

The first class is the base exception class, while the class underflow is a *derived* class or sub-class of the exception class. The syntax for derived classes in C++ is similar to that in C#, whereas Java uses a keyword, for example

```
public class Underflow extends Exception { … }
```

In order to throw such an exception, we must include the exception header file in our `Array.h` file, as well:

```
#include "Exception.h"
```

We should this include this immediately below the inclusion of the library file `algorithm`. (In general, it is usual to list the library header files that are included first, followed by other header files.) The mechanism to throw an instance of an exception is to create one calling the constructor and then *throwing* that instance:

```
int Array::average() const {
    if ( empty() ) {
        throw underflow();
    }

    // find the average
}
```

The constructor for the underflow exception *could* take additional arguments; for example, you *could* (we do not) update the constructor to take a string that described the issue at hand:

```
int Array::average() const {
    if ( empty() ) {
        // We are not doing this; this is for demonstration only
        throw underflow( "trying to find the average of an empty array" );
    }

    // find the average
}
```

In C, there were no exceptions: there were two mechanisms to signal an exception, including

1. calling `exit( int )` which exits the program, or
2. using a combination of `int setjmp( jmp_buf )` and `void longjmp( jmp_buf, int )`.

It is also to redirect print statements to standard error instead of standard output. If you are interested in the use of `setjmp` and `longjmp`, the corresponding Wikipedia entry is quite reasonable.

Exceptions are *thrown*, and therefore they can also be *caught*. Suppose we have a function like the following:

```
#include "Array.h"

int main() {
    double ave;
    Array info( 10 );

    try {
        ave = info.average();
    } catch ( underflow excpt ) {
        // the array 'info' is empty--use a default value
        ave = 0.0;
    }

    return 0;
}
```

If, during a call to the `average` function, an exception is thrown, the type of the exception thrown is matched by the one catch that appears (there can be more than one catch). Inside this block, the local variable `excpt` will be assigned the instance of the class that was thrown.

Now that we have dealt with the special case, let us go on. Your first thought might be to implement the average member function as

```
double Array::average() const {
    if ( empty() ) {
            throw underflow();
    }

    // The average is the sum of the entries divided by the size
    return sum()/size();
}
```

If you try implementing the function

```
double f() {
    return 1;
}
```

the compiler will determine that the result is an integer, but that it must be converted to a double-precision floating-point number. Thus, rather than returning `0x00000001`, it will return `0x3ff0000000000000`. Similarly, the result of the calculation `sum()/size()` will be converted into a double-precision floating-point number; however, the compiler will see the division and determine that:

1. the numerator is an `int`, and
2. the denominator is an `int`.

Therefore, it will call an integer division instruction. For example, if the array held the entries 1, 2, 3 and 4, the sum would be 10 and the size would be 4, and `10/4` will evaluate to `2` (discarding the remainder).

Instead, we need a mechanism to tell the compiler that we really want to divide two double-precision floating-point numbers. The only way to do this is to *cast* the operands to be of type `double`:

```
return static_cast<double>( sum() )/static_cast<double>( size() );
```

Whatever the values of sum() and size() are, these are to be converted to (or *cast as*) double-precision floating-point numbers. Now, the compiler will see that the numerator and denominator are of type `double`, and therefore it will call a floating-point division instruction. Thus, our previous example would therefore evaluate to 2.5, as expected.

In implementing the variance function, your may try to implement it as:

```
double Array::variance() const {
    if ( size() <= 1 ) {
        throw underflow();
    }

    double av = average();

    // sum of the squares of the differences
    double ssdiff = 0.0;

    for ( int i = 0; i < size(); ++i ) {
        ssdiff += (internal_array[i] - av)^2;
    }

    return ssdiff/(size() - 1);
}
```

In this case, because `ssdiff` is already of type `double`, the compiler sees the return statement and determines:

1. the numerator is a `double`, and
2. the denominator is an `int`.

Therefore, the only logical approach is to convert the denominator into a `double` and call a floating-point division instruction. There is, however, a different problem: in many cases, we think of the caret character (^) as an exponentiation operator (certainly, that is how Matlab interprets it). In C and C++, however, it is the bit-wise exclusive-or (xor) operator. Thus, we must modify the line

```
ssdiff += (internal_array[i] - av)^2;
```

to

```
ssdiff += (internal_array[i] - av)*(internal_array[i] - av);
```

Fortunately, most compilers would not calculate these two differences twice; instead, they would calculate the difference once and then multiply the result by itself.

Finally, the sample standard deviation member function need only calculate the square root of the

```
double Array::std_dev() const {
    return std::sqrt( variance() );
}
```

To use the square-root function, we must include the `cmath` library along with the `algorithm` library.

```
#include <algorithm>
#include <cmath>
#include "Exception.h"
```

If the size is 0 or 1, the variance function will throw the appropriate exception, so there is no point in throwing an exception from this function.

In all your projects, always ask if you can implement one function using simpler functions you have previously implemented.

0.11 Memory deallocation

Suppose we have a function that declares local variable to be of type `Array`:

```
void f() {
    Array local_array( 100 );
    // does something
}
```

Similarly, suppose create a new instance of the array class, and then later delete it:

```
Array *array_ptr = new Array( 100 );

// does something

delete array_ptr;
```

For these:

1. In the first case, the memory for the three member variables for the capacity, the internal array and the size are allocated by the compiler, and the constructor calls for an array of 100 integers (400 bytes).
2. In the second case, the request to `new` sees the operating system allocate memory for the three member variables, and the constructor then makes a subsequent call to for an array of 100 integers.

By default, however,

1. in the first case, when the local variable `local_array` goes out of scope, the compiler knows to deallocate the memory for the three member variables, but will not deallocate the 400 bytes, and
2. in the second, when `delete` is called, the memory for the three member variables is returned to the operating system, but not the additional memory for the 400 bytes.

This is because the compiler does not know what you want to do with additional memory allocated during the call to the constructor—maybe you wanted to use that elsewhere. Consequently, the 400 bytes is lost, because the only variable storing the address of those 400 bytes has been deallocated. This forms a *memory leak*. To fix this, we need some way of saying:

“When we are deallocating the memory for this object,
we need to deallocate the memory for the array, as well.

We can define this in a member function called the *destructor*, which is given the declaration `~Array()`:

```
class Array {
    private:
        int array_capacity;
        int *internal_array;
        int array_size;
    public:
        Array( int = 10 );
        ~Array();

        // other member functions
};
```

Recall that `~` is the bit-wise not operator in C++ and that `Array()` is the default constructor, so `~Array()` is the destructor (or *not the constructor*). In the destructor, we don't have to worry about the memory allocated for the member variables, we only have to worry about the additional memory allocated in the constructor:

```
// Destructor
//  - deallocate the memory for the array

Array::~Array() {
    delete [] internal_array;
}
```

Note that if you used `new` to allocate a single instance of a class, such as

```
    Class *ptr = new Class( args... );
```

the call to `delete` must be

```
    delete ptr;
```

However, if you used `new` to allocate an array of instances of a class, such as

```
    Class *ptr = new Class[ARRAY_SIZE];
```

the call to delete must also flag that it is an array of objects being deleted:

```
    delete [] ptr;
```

0.12 Accessing entries

We can currently append items into this array and query descriptions of that data, but we cannot access individual entries. Consequently, it might be nice to include a function that tells us what is *at* a particular location:

```
int Array::at( int n ) const {
    if ( n < 0 || n >= size() ) {
        throw out_of_range();
    }

    return internal_array[n];
}
```

Note that this function checks to ensure that the argument falls within the range of entries, and if it does not, it throws an illegal argument exception. We can now print the entries:

```
Array info( 20 );
// Add some entries...

// Print the entries
for ( int i = 0; i < info.size(); ++i ) {
    cout << info.at( i ) << " " << endl;
}
```

It would be really nice if we could, instead, use a more natural notation, such as:

```
// Print the entries
for ( int i = 0; i < info.size(); ++i ) {
    cout << info[i] << " " << endl;
}
```

however, `info` is not a pointer, it is an instance of the `Array` class. Fortunately, the indexing operator is simply that: an operator no different than unary `*` which dereferences a pointer.

Note that for a pointer, `array[n]` is identical to `*(array + n)`—both access the n^{th} entry of the array. Note that if you add an integer to a pointer, the compiler determines the type of the pointer and steps forward that many bytes for each value, so these two are quite literally identical.

C++ allows you to overload most operators (including `new` and `delete`, but not, for example, `.`).

Suppose that ☺ is a binary C++ operator. For example, you may have the operation `a ☺ b`. If you want to override this operator where a is an instance of the type `LHS_type`, you must implement a function

```
return_type LHS_type::operator☺( RHS_type const & );
```

If you want to override the indexing operator to have `a[b]` interpreted as something other you must implement the function

```
return_type A_type::operator[]( B_type const & );
```

or, if the type of the index is an integer, you would use

```
return_type A_type::operator[]( int );
```

Note that operator overloading is very powerful, and it can in many cases make it possible to write very clean and succinct programs.

In this case, we could therefore define a function

```
int Array::operator[]( int n ) const {
    return internal_array[n];
}
```

Thus, when you execute the following code:

```
int main() {
    Array info( 10 );

    info.append( 42 );
    info.append( 64 );

    std::cout << info[0] << std::endl;

    return 0;
}
```

the following happens:

1. The compiler determines with `info[0]` that `info` is of type `Array`.
2. The class `Array` has a member function `operator[]( int ) const`.
3. The index `0` can be interpreted as an `int`, so the overloaded operator function is called with the parameter `n` assigned the value of the argument `0`.
4. Inside the `operator[]` function, the compiler determines that the member variable `interal_array` is of type `int*`, and therefore `internal_array[0]` is simply a request for the value stored in the first entry of the array starting at the address stored in `internal_array`.

Note that we have two functions that perform the exact same operation: `at(…)` and `operator[](…)`. The Standard Template Library (STL) has the same: every container class implements these two functions, with the following difference:
1. The `at(…)` member function checks to ensure that the index does not fall outside the range of the container, and if it does, an `out_of_range` exception will be thrown.
2. The `operator[](…)` does not check the range; instead, it will simply try

The first function is always slower than the second, and the second should only be used if the programmer is sure that the index is within the appropriate range.

Humorously, because `operator[]` is a member function, it is also possible, but probably undesirable, to call it directly: `info.operator[]( 0 )` instead of `info[0]`.

0.13 Further functionality

We will now consider three additional problems:

1. How do we swap to instances of our array class?
2. How do we copy an instance of our array class?
3. How do we assign an instance of our array class to another?

For example,

```
Array a( 3 ), b( 5 );
b.append( 52 );
a.swap( b );

Array a( 5 );
a.append( 35 );
a.append( 42 );
Array b( a );  // make b a copy of 'a'

Array a( 5 );
a.append( 35 );
Array b( 7 );
b.append( 42 );
a = b;
```

Another place we may want to make a copy is if we make a pass-by-value or a return-by-value in a function call. For example, if we define the function

```
void f( Array param ) {
    // 'param' is a copy of the argument--it is passed by value
    // Do something with 'param'
}
```

If we now create an array and then pass it to this function, it is passed-by-value, and therefore a copy of the array must be made.

0.13.1 Swapping two instances

To swap one instance of an array with another, all that is necessary is that the three member variables be swapped. To swap instances of primitive data types, you can use the `std::swap` function implemented in the `algorithm` library.

```
void Array::swap( Array &other ) {
    std::swap( array_capacity, other.array_capacity );
    std::swap( internal_array, other.internal_array );
    std::swap( array_size,     other.array_size );
}
```

Note that the argument is passed by reference—we want to swap this instance of the class with the instance passed as the argument.

0.13.2 Making a copy

The default behavior for making a copy of an object is to simply copy all the member variables from the original to the copy, as is done when an argument is passed by value to a function. In the case of a simple complex-number class, this might be acceptable, as just copying the real and imaginary parts of the complex number would make a copy. For our array class, however, it isn't so easy. Consider what happens if we have the following class:

```
int main() {
    Array first( 6 );
    first.append(  13 );
    first.append(  17 );
    first.append(  39 );
    first.append( 666 );

    f( first );

    cout << first[0];
    return 0;
}

void f( Array second ) {
    // do something
}
```

When the first array is created, memory is allocated for the array of capacity six, and therefore we now have

```
first                                   0x000b3820          13
array_capacity          6               0x000b3824          17
internal_array  0x000b3820              0x000b3828          39
array_size              4               0x000b382c         666
                                        0x000b3830           ?
                                        0x000b3834           ?
```

When the instance first is passed by value, by default, the values of the member variables are simply copied over, creating a parameter `second`.

```
second

array_capacity          6
internal_array  0x000b3820
array_size              4
```

Note that both `first` and `second` store the address of the same array. The problem comes when the function `f` returns—it calls the destructor on the parameter, and the destructor calls

```
delete [] internal_array;
```

Thus, once we are back in `main()`, `first.internal_array` still stores the address `0x000b3820`, but this address is no longer valid—the array at this address was deallocated.

Such a copy which simply copies over the values of the member variables is called a *shallow copy*. In this case, however, what we really need to do is create an entirely new array for our copy, and the values in the array of the argument must be copied over to the copy. Thus,

```
second
                                       0x00176a48       13
array_capacity          6              0x00176a4c       17
internal_array 0x00176a48              0x00176a50       39
array_size              4              0x00176a54      666
                                       0x00176a58        ?
                                       0x00176a5c        ?
```

Now, when the destructor is called on the parameter `second`, the new array is deallocated.

This is implemented in C++ (and many other object-oriented languages) as a *copy constructor*, as a new object is being made, but rather than using the constructor we have defined, the constructor is making a copy of an already existing object. Like the constructor we have already defined, the copy constructor allows you to initialize the variables. The following is an actual implementation of the default shallow copy constructor:

```
Array::Array( Array const &other ):
array_capacity( other.array_capacity ),
internal_array( other.internal_array ),
array_size( other.array_size ) {
    // empty
}
```

Note that the values of the three member variables of the argument are assigned to the three member variables of the new copy. Unfortunately, we require more: first, we must create a new array, but then we must also copy the old values over:

```
Array::Array( Array const &other ):
array_capacity( other.array_capacity ),
internal_array( new int[array_capacity] ),
array_size( other.array_size ) {
    // copy over the values from one array to the other
    for ( int i = 0; i < size(); ++i ) {
        internal_array[i] = other.internal_array[i];
    }
}
```

Notice that we don't have to check that `other.array_capacity > 0`, as that would have already been checked when the constructor was initially called on that object being copied.

At this point, it should be noted that a pass-by-value can therefore be very expensive: a large object passed by value may require a significant amount of time to copy, and it may require a significant amount of additional memory. For this reason, there are three options for passing objects:

1. pass-by-value,
2. pass-by-reference, and
3. pass-by-constant-reference.

The first calls the copy constructor to create an instance of the class, and when the function finishes execution, it calls the destructor on the parameter. In this case, the 42 is appended to the copy, but does not change the original.

```
int init( Array data, ... ) {
    // The first argument is passed-by-value and a copy of that argument is
    // assigned to the parameter 'data', and because a copy constructor is
    // declared, it will be initialized with the first argument of init(...)
    // passed as the argument to the copy constructor does something...

    data.append( 42 );

    return 0;
    // Just before the function returns, the destructor is called on 'data'
}
```

Pass-by-reference does not make a copy, and therefore it does not call the destructor, either. In the following function, the 42 is appended to the original argument.

```
int init( Array &data, ... ) {
    // Here, the parameter 'data' refers to the original first argument passed
    // to the init(...) function

    // Append 42 to the original argument
    data.append( 42 );

    return 0;
    // No destructor is called on 'data'
    //     the original continues to exist
}
```

Under some conditions, the problem with pass-by-reference is exactly the problem solved by pass-by-value: pass-by-reference allows the function to change the object in question. This may not be the intention of the programmer, and therefore pass-by-constant-reference prevents the function from changing the class. If the parameter is declared to be a constant reference, the function can only call those member functions declared to be read-only (or `const`).

```
int init( Array const &data, ... ) {
    // Here, the parameter 'data' refers to the original first argument passed
    // to the init(...) function

    // We can call data.size(), data.sum(), data.average(), etc.,
    // but we cannot call data.append(…) or data.clear()
    double av = data.average();
    double std = data.std_dev()/std::sqrt( size() );

    std::cout << "A 95% confidence interval for the the mean is ["
              << (av - 1.96*std) << ", " << (av + 1.96*std) << "]"
              << std::endl;

    return 0;
    // No destructor is called on 'data'
    //     the original continues to exist
}
```

If you try to insert a call to `append(…)` in this function, the compiler will issue an error:

```
test.cpp: In function 'int init(const Array&,...)':
test.cpp:17: error: passing 'const Array' as 'this' argument of
                    'bool Array::append(int)' discards qualifiers
```

0.13.3 Assigning an object

Suppose we have two instances of our array class, `a` and `b`, and we then assign one to the other:

```
a = b;
```

Similar to the copy constructor, the default behavior is to make a shallow copy of the right-hand side: the member variables of the instance `a` are assigned the values of the variables in instance `b`. As with making a copy, this is inappropriate for our class, as now both `a.internal_array` and `b.internal_array` would store the same address. In addition, the old value of `a.internal_array` is now lost, so there is no way to deallocate the memory associated with that array—we have a memory leak.

To override the default assignment operation, we must override the `operator=` function. Without going into details, we will look at the standard solution:

```
Array &Array::operator=( Array rhs ) {
    swap( rhs );
    return *this;
}
```

Thus, if you call `a = b`, the right-hand side is passed-by-value and copied to the parameter `rhs`. The function is called on the left-hand operand, and therefore `this` is assigned the address of `a`. The way that this works is as follows. Suppose we execute the following code:

```
Array a( 3 );
a.append( 35 );
a.append( 75 );
Array b( 5 );
b.append( 42 );
a = b;
```

Now our space looks as follows: Memory for two local variables has been allocated and these store the addresses of two arrays stored in other locations in memory.

```
a

array_capacity            3        0x000b3820          35
internal_array   0x000b3820        0x000b3824          75
array_size                2        0x000b3828           ?


b
                                    0x00176a48          42
array_capacity            5         0x00176a4c           ?
internal_array   0x00176a48         0x00176a50           ?
array_size                1         0x00176a54           ?
                                    0x00176a58           ?
```

When the assignment operator function is called, the argument `b` is passed-by-value, so the copy constructor is called and a deep copy is made and assigned to the parameter `rhs`.

```
rhs
                                    0x005b9a04          42
array_capacity            5         0x005b9a08           ?
internal_array   0x005b9a04         0x005b9a0c           ?
array_size                1         0x005b9a10           ?
                                    0x005b9a14           ?
```

The first statement in the assignment operator function is a call to swap. This swaps the entries of **rhs** and the object the member function is called on: **a**. Now that these are swapped, they resemble

```
a

array_capacity          5        0x000b3820          35
internal_array 0x005b9a04        0x000b3824          75
array_size              1        0x000b3828           ?
```

```
rhs
                                  0x005b9a04          42
array_capacity          3         0x005b9a08           ?
internal_array 0x000b3820         0x005b9a0c           ?
array_size              2         0x005b9a10           ?
                                  0x005b9a14           ?
```

Notice that the address stored in **a**.internal_array is now the address of the newly copied array, while **rhs**.internal_array now points to the original array. It is necessary that the assignment operator return a reference to the object on the left-hand side, and therefore we return `*this`. Once the function exits, because the parameter was passed-by-value, the destructor will be called on **rhs**. This will call delete on the memory address **0x000b3820**, which is the address of the original array for **a**.

This is the recommended means of writing a default deep assignment operator. Please note that there are times where this is unnecessarily expensive, and therefore it may be desirable to write a more efficient implementation. For example, suppose that both the left- and right-hand sides already had arrays that were of the same capacity. In this case, creating a brand new array might unnecessary. To accomplish this, we must use a pass-by—constant-reference, but we must now also be aware that the user may accidently assign an object to itself—in which case, we do nothing.

```
    Array &Array::operator=( Array const &rhs ) {
        if ( this == &rhs ) {
            // Do nothing if an object is assigned to itself
        } else if ( capacity() == rhs.capacity() ) {
            // If both objects have arrays with the same capacity
            array_size = rhs.size();

            for ( int i = 0; i < size(); ++i ) {
                internal_array[i] = rhs.internal_array[i];
            }
        } else {
            // If the capacities are different, call the copy constructor
            // to create a copy of rhs and then swap the copy and this.
            Array tmp( rhs );
            swap( tmp );
        }

        return *this;
    }
```

In this class, we will always use the recommended default assignment operator.

If you are wondering why the return value is `Array &` (a reference to the array), this is necessary if you have, for example, a chain of assignments. Consider the statement

```
int a = 4, b = 7, c = 9;
a = b = c;
```

The assignment operator is right-associative, so the compiler interprets the second statement as

```
a = (b = c);
```

Thus, **`b`** is assigned the value of **`c`**, and the return value is the new value of **`b`**. Next, **`a`** is assigned that returned value, and therefore **`a`** now also has the new value of **`b`**.

0.13.4 Summary of three container operations

Here we have considered how to swap two instances of our array class and how to implement a deep copy constructor. The recommended default assignment operator uses both of these.

0.14 Printing our array and friendship

Suppose we try to print our array:

```
Array a( 5 );
a.append( 35 );
a.append( 42 );
std::cout << a;
```

When the compiler sees the statement `std::cout << a`, it determines that:

1. the left-hand side is of type `ostream`, and
2. the right-hand side is of type `Array`.

At this point, the compiler notes there is no overloaded function of the left-shift operator that takes an `ostream` object as a first argument and an `Array` object as a second. We must therefore define such a function. The following function prints the entries of the array

```
std::ostream &operator<<( std::ostream &out, Array const &para ) {
    if ( para.empty() ) {
        out << "-";
    } else {
       out << para[0];
    }

    for ( int i = 1; i < para.size(); ++i ) {
        out << " " << para[i];
    }
    for ( int i = para.size(); i < para.capacity(); ++i ) {
        out << " -";
    }
    return out;
}
```

For example, if an array of capacity 10 holds the three entries 1, 2 and 4, the output would be equivalent to printing the string

```
"1 2 4 - - - - - - -"
```

Note that this is not a member function of the Array class, it is simply a function that takes two arguments: the first is of type `ostream` and the second is of type `Array`. It returns the `ostream` argument so that you can perform chained printing statements, such as

```
std::cout << a << std::endl;
```

Because this function only calls the public member functions of the `Array` class, we can print the array without any issues; however, what happens if, in order to print your class, you need access to private member variables? Suppose, for example, you could not access the capacity of the array through a public member function. In this case, it is possible for the Array class to declare this function to be a *friend*; that is, it may indicate to the compiler that this function is allowed to access the private member variables and call the private member functions of the Array class. In the class definition, this requires the following statement:

```
class Array {
    private:
        int array_capacity;
        int *internal_array;
        int array_size;

    public:
        Array( int = 10 );
        Array( Array const & );
        ~Array();

        // Accessors and mutators...

    // Friends
    friend std::ostream &operator<<( std::ostream &, Array const & );
};
```

Thus, we could avoid calling the indexing operator and access the internal array directly:

```
std::ostream &operator<<( std::ostream &out, Array const &para ) {
    if ( para.empty() ) {
        out << "-";
    } else {
       out << para.internal_array[0];
    }

    for ( int i = 1; i < para.size(); ++i ) {
        out << " " << para.internal_array[i];
    }
    for ( int i = para.size(); i < para.capacity(); ++i ) {
        out << " -";
    }
    return out;
}
```

If this function was not declared a friend of the class `Array`, this would not be possible.

0.15 How about arrays of doubles (templates)?

Suppose, having completed the array class, you suddenly determine you need a similar class, but now it is necessary that it stores double-representation floating-point numbers. You may, initially consider doing a search-and-replace, replacing each instance of `int` with `double`; however, this—you quickly realize—is futile. Such a blind search and replacement would cause numerous faults:

```
// A friend to prdouble the array
 friend std::ostream &operator<<( std::ostream &, Array const & );

double Array::size() const {
    return array_size;
}

double Array::variance() const {
    if ( size() <= 1 ) {
        throw underflow();
    }
    double av = average();
    double ssdiff = 0;
    for ( double i = 0; i < size(); ++i ) {
        ssdiff += (internal_array[i] - av)*(internal_array[i] - av);
    }
    return ssdiff/(size() - 1);
}
```

Suddenly, `size()` is returning a `double`, the repetition statement (for-loop) is iterating over a `double`, and you are no longer pr*int*ing an array. At this point, you may come up with a second idea:

> Just replace all type `int` that refer to entries in the array with an arbitrary symbol, say **`Type`**, and then do a search-and-replace of **`Type`** with `int` (call that file `Array_int.h`) and then do a second search-and-replace of **`Type`** with `double` (call that file `Array_double.h`)

Thus, our class now looks like the following:

```
class Array {
    private:
        int array_capacity;
        Type *internal_array;
        int array_size;
    public:
        Array( int = 10 );

        // Accessors
        int size() const;
        int capacity() const;
        bool empty() const;
        bool full() const;
        Type sum() const;
        Type prod() const;
        Type max() const;
        Type min() const;
        double average() const;
        double variance() const;
        double std_dev() const;
        Type at( int ) const;
        Type operator[]( int );

        // Mutators
        void append( Type const & );
        void clear();
};

Array::Array( int n ):
array_capacity( std::max( 1, n ) ),
internal_array( new Type[array_capacity] ),
array_size( 0 ) {
    // does nothing
}

bool Array::append( Type const &obj ) {
    if ( full() ) {
        return false;
    }

    // currently, entries 0, ..., array_size – 1
    // are occupied
    internal_array[array_size] = obj;
    ++array_size;
    return true;
}
```

Now you ask yourself: wait if I can do this with a search-and-replace, can't C++ do this for me? Fortunately, C++ can, and it does so using a mechanism called *templates*.

Given any class, function or member function, it is possible to indicate that specific symbols can be replaced by the user.

```
template <typename T>
void my_bad_swap( T &x, T &y ) {
    // Don't do this...this is inefficient but shown as an example
    T tmp(x);
    x = y;
    y = tmp;
}
```

Now this function can be used to swap any types:

```
int i = 3, j = 4;
double x = 3.14, y = 6.18;
Array a(3), b(7);

swap<int>( i, j )
swap<double>( x, y );
swap<Array>( a, b );
```

As an aside, if you just called `swap( i, j )` or `swap( a, b )`, the compiler will determine that the implied type is `int` and `Array`, respectively.

Thus, our array class now looks like

```
template <typename Type>
class Array {
    private:
        int array_capacity;
        Type *internal_array;
        int array_size;
    public:
        Array( int = 10 );

        // Accessors
        int size() const;
        int capacity() const;
        bool empty() const;
        bool full() const;
        Type sum() const;
        Type prod() const;
        // and so on...
};
```

The `template <typename Type>` statement modifies the next structure, be it a function or class. Consequently, it is necessary to prefix each member function; for example:

```
template <typename Type>
Array<Type>::Array( int n ):
array_capacity( std::max( 1, n ) ),
internal_array( new Type[array_capacity] ),
array_size( 0 ) {
    // does nothing
}

template <typename Type>
int Array<Type>::size() const {
    return array_size;
}

template <typename Type>
bool Array<Type>::append( Type const &obj ) {
    if ( full() ) {
        return false;
    }

    // currently, entries 0, ..., array_size - 1
    // are occupied
    internal_array[array_size] = obj;
    ++array_size;
    return true;
}
```

At this point, we can declare an array of integers and an array of doubles:

```
#include <iostream>
#include "Array.h"

int main() {
    // Create an array of 10 ints
    Array<int> info( 10 );

    info.append( 42 );
    info.append( 91 );
    info.append( 35 );

    // Prints "42 91 35 - - - - - - -"
    std::cout << info << std::endl;

    // Create an array of 6 doubles
    Array<double> data( 6 );
    data.append( 42.52 );
    data.append( 91.41 );
    data.append( 35.91 );
    data.append( 83.19 );

    // Prints "42.52 91.41 35.91 83.19 - -"
    std::cout << data << std::endl;

    return 0;
}
```

0.16 Testing

As you are writing your class, you should also be writing tests to ensure that the implementation is working correctly. For example, you could write one test as follows:

```
#include <iostream>
#include <cassert>
#include "Array.h"

int main() {
    // Create an array of 10 ints
    Array<int> a;

    assert( a.capacity() == 10 );
    assert( a.empty() == true );

    for ( int i = 0; i < 10; ++i ) {
        assert( a.size() == i );
        assert( a.full() == false );
        assert( a.append( i ) == true );
        assert( a.empty() == false );
        assert( a.at( i ) == i );
        assert( a[i] == i );
    }

    assert( a.size() == 10 );
    assert( a.full() == true );
    assert( a.append( 10 ) == false );

    Array<int> b( 4 );

    assert( b.capacity() == 4 );
    assert( b.empty() == true );

    for ( int i = 0; i < 4; ++i ) {
        assert( b.full() == false );
        assert( b.append( i ) == 1 );
        assert( b.empty() == false );
    }

    assert( b.size() == 4 );
    assert( b.full() == true );
    assert( b.append( 4 ) == 0 );

    Array<int> c( -1 );
    assert( c.capacity() == 1 );
    assert( c.size() == 0 );
    assert( c.empty() == true );
    assert( c.full() == false );
    assert( c.append( 0 ) == true );
    assert( c.append( 1 ) == false );
    // And so on...

    return 0;
}
```

This test numerous aspects of the instance of the class, but not all. Additionally, there are tools that you can use to develop a more comprehensive suite of tests. We, however, will be using a testing environment that also allows you to interactively work with your class. We have developed a `Tester` class that creates an interface that allows you to interact with the data structure you have written. For each data structure, there is a specialized derived class from this `Tester` class. The naming convention is that the derived class is called `Array_tester`. There is a file `Array_driver.cpp` which includes a `main()` function that creates an instance of, for example, one of

1. `Array_tester<short>`
2. `Array_tester<int>`
3. `Array_tester<float>`
4. `Array_tester<double>`

The tester class has a public member function `run()` which then creates an interactive environment for working with your class. (Fortunately, all this is done for you.)

You could recreate the previous executable by executing the following:

```
$ ./a.out int
$ new
$ capacity 10
$ empty 1
$ size 0
$ full 0
$ append 10 1
$ empty 0
$ append 11 1
$ append 12 1
$ append 13 1
$ append 14 1
$ append 15 1
$ append 16 1
$ append 17 1
$ append 18 1
$ full 0
$ append 19 1
$ full 1
$ size 10
$ capacity 10
$ append 20 0
$ delete
$ new: 4
$ capacity 4
$ empty 1
$ full 0
$ append 30 1
$ append 31 1
$ append 32 1
$ full 0
$ append 33 1
$ full 1
$ size 4
$ capacity 4
$ append 34 0
$ delete
$ new: -1
$ capacity 1
$ size 0
$ empty 1
$ full 0
$ append 40 1
$ size 1
$ empty 0
$ full 1
$ delete
$ details
$ exit
```

At each step, you will see appropriate output. One problem, however, is that you will quickly determine that there is a lot of additional work required here, and it would be nice if this could be automated.

You can take all these commands and create a text file, say `test.in.txt`, and include these commands in that file. You can include C++ end-of-line comments and empty lines to help explain your tests.

```
// Create an array with the default size (10)
new
capacity 10
empty 1
size 0
full 0
append 10 1
empty 0
append 11 1
append 12 1
append 13 1
append 14 1
append 15 1
append 16 1
append 17 1
append 18 1
full 0
append 19 1
full 1
size 10
capacity 10
append 20 0
delete

// Create an array with capacity 4
new: 4
capacity 4
empty 1
full 0
append 30 1
append 31 1
append 32 1
full 0
append 33 1
full 1
size 4
capacity 4
append 34 0
delete

// Create an array of with the constructor being passed -1
new: -1
capacity 1
size 0
empty 1
full 0
append 40 1
size 1
empty 0
full 1
delete
details
exit
```

Now, at the command prompt, you can execute

```
$ ./a.out int < test.in.txt
```

0.17 Summary

In this laboratory, we have covered:

1. laboratory safety,
2. an overview of the project requirements,
3. a review of pass-by-value versus pass-by-reference,
4. seen how to log into Unix and the introduction of Unix commands,
5. the *Hello world!* program,
6. how to create a project in both Unix and Visual Studio,
7. the Array.h header file,
8. improvements to our implementation,
9. additional functionality to our class,
10. four statistical functions,
11. memory deallocation,
12. accessing entries in the array,
13. discussing further functionality, including:
    a. swapping two instances of our class,
    b. making a copy of an instance, and
    c. assigning an instance of our class to a variable already holding the value,
14. printing our array and friendship,
15. templates, and
16. testing our class.

At this point, you should be ready for Project 1.