Instant C++ Programming
CS A331

This document assumes that you are familiar with Java and primarily highlights the differences between the two languages.

First, here is the general format for a C++ program:

File: program.cpp

```
/* Comments */
// More comments
#include <iostream>
#include <string>
…                 // Provides access to standard code libraries

#include "myclass.h"    // Provides access to user-defined code

int g;                  // Global variables defined here

using namespace std;    // Defines the namespace we're using

// Define any functions here that are available globally, e.g.
void someFunc() {
      // code
}

int main(char *argv[], int argc)          // Parameters optional
{
      // Code here for main returning an int value
}
```

If we would like to add classes, then each class is generally created in two files. The .h file contains the header information, while the .cpp file contains the actual code. The template would like the following for a class named "myclass":

File: myclass.h

```
#include …                    // Any includes up here
class myclass
{
      public:
            myclass();        // Default constructor
            ~myclass();       // Destructor
            void method();    // Define our methods here
            int method2(int i); // Defines the interface
                                // to each method
            double d;         // Define any public variables
      private:
            int x;            // Define any private variables
            void method3();   // Define any private methods
};                            // Don't forget the semicolon here
```

File: myclass.cpp:

```
#include "myclass.h"     // Any includes up here
using namespace std;

myclass::myclass()       // class name followed by ::
                         // followed by method name
{
      // Code for constructor of this class
}

myclass::~myclass()
{
      // Code for destructor,
      // invoked when this class is destroyed
}

void myclass::method1()
{
      // Code for method 1
}

int myclass::method2()
{
      // code for method 2, returns an int
}

void myclass::method3()
{
      // code for method 3
}
```

myclass.cpp is called the implementation file and is compiled separately from the header file, myclass.h. If myclass.h is going to be included in multiple implementation files, then there is a problem redefining the class. Use the #ifndef directive to only define it once:

```
#ifndef _MYCLASS_H
#define _MYCLASS_H
#include …                    // Any includes up here
class myclass
{
      public:
            myclass();         // Default constructor
            ~myclass();        // Destructor
            void method();     // Define our methods here
            int method2(int i); // Defines the interface
                                 // to each method
            double d;          // Define any public variables
      private:
            int x;             // Define any private variables
            void method3();    // Define any private methods
};
#endif
```

If we would like to have a class be derived from another class (i.e. inherit the properties of some superclass) we can denote this in the .h file when we define the class. For example, the following defines a class called Auto that inherits from the class called Vehicle:

```
class Auto :: public Vehicle
{
        // Definition of Auto class continues here
}
```

Inheritance works very similarly to Java. A subclass inherits all public variables and methods of the superclass. Private data, constructors, destructors, and operators are not inherited. If a subclass has the same name as a method in the parent class, the method in the subclass overrides the parent method. When a subclass is destroyed, upon destruction the destructor of the superclass will be invoked.

In Java, you can use the super keyword to access the superclass method of an object in the event that the subclass and superclass have the same method name. You can use the name resolution operator to the same in C++. For example, say that both class Auto and class Vehicle have a "GetID()" method.

The code:

```
Auto a;                 // Note we don't need a "new" unless we're using pointers
                        // Will allocate space for Auto on the stack as a local var
a.GetID();              // Invokes GetID() from the Auto class
a.Vehicle::GetID();     // Invokes GetID() from the Vehicle class
```

To compile C++ files in Unix, you can use the command:

```
g++ file.cpp            (or g++ *.c  to compile everything)
```

This creates a file named "a.out" that you can then execute. In an IDE environment such as Visual Studio you will need to create a new project, add the source files to it, and then build it. We will give a demonstration in class.

C++ Basics

Now that we have seen the layout of a C++ program, let's see some basic innards that make up a typical C++ program.

Primitive Data Types: Just like Java.

char, int, long, float, double, bool (not boolean)

C++ treats 0 as false and non-zero as true

Strings: Not quite like Java.
C++ has no built-in data type that behaves like a Java string object. Instead, core C++ strings are defined as arrays of char. The null character \0 is used to terminate a string. C++ does include a Standard Template Library (STL) implementation of a "string" data type that behaves similarly to a Java string. (The STL library includes common data structures, such as a vector, hash table, etc.)

To use the STL strings, we must #include <string> at the top of each file that wishes to use strings. Here is an example:

```
#include <string>
using namespace std;
int main()
{
  string s;       // Note lowercase string, not String
  s = "hello";
  s = "hello" + "there";      // Concatenation
  s.length();                 // Returns # of chars in the string
  s[0];                       // References character 'h'
  s[1];                       // References character 'e'
  return 0;
}
```

Output: To print data to the console, use:

```
cout <<  data_to_print;
```

For example:

```
cout << "hello world";              // Outputs hello world
cout << s << " " << x+3;            // Outputs string s concatenated with x+3
```

To print a newline at the end of the output use endl:

```
cout << "hello world" << endl;
```

We can also include the various escape characters (\n, \r, \", etc.) in the string.

Input: To read data from the keyboard, use cin. For example:

```
int x;
double d;
cin >> x;                   // Reads an integer from the keyboard into x
cout << x << endl;          // Output what the user entered
cin >> x >> d;              // Input into multiple variables
```

If we are inputting into a string, this doesn't quite behave as expected if there are spaces in the input. cin only inputs data to the first whitespace, so if we have input with spaces we don't get the desired result:

```
string s;
cin >> s;                  // User types "hello world"
cout << s << endl;         // Outputs "hello"
cin >> s;                  // Reads in the "world" part
cout << s << endl;         // Outputs "world"
```

To avoid this problem we can use various functions such as getline() to input a string to the newline, but we won't really deal with this issue in this short introduction to C++.

Boolean expressions, arithmetic operators, relational operators: Just like Java

If statement: Just like Java.
One exception is that any non-zero value is considered to be true, while zero is considered to be false. So we could make a statement such as:

```
if (1) { … }   which amounts to :  if (true) { … }
```

Assignment statement: Just like Java
One common pitfall is confusing = with ==. While Java will flag this as an error, C++ will not because it is considered legal. An example is below:

```
int i=0;
if (i=1) { …  }
```

The body of the if statement will always be executed because in the expression "i=1" we assign the value 1 into i, and the value tested by the if statement amounts to if (1) {…} As we have seen this is considered true, so we will execute the boy of the statement and at the same time variable i is set to 1.

For loop, while loop, do-while loop, break, continue: Just like Java

Defining methods: Call by value works just like Java. We must define the method prototype in the .h file and then place the actual code for the method in the .cpp implementation file.

One difference is that C++ allows us to optionally pass a parameter by reference. In this mode, the address of the variable is passed to the method instead of putting a copy of the value on the stack. By having the address of the source variable, changes to the variable in the method result in changes to the variable in the caller. To declare a parameter as reference, put an & in front of the variable name in the method prototype. For example:

```
void ChangeValues(int &y)
{
        y = 20;
        return;
}

int main()
{
        int x = 3;
        ChangeValues(x);
        cout << x << endl;              // Outputs 20
        return 0;
}
```

If ChangeValues was defined as void ChangeValues(int y) then we get the Call by Value behavior and x retains the value 3 inside main.

Scoping: Same as Java. Variables defined in methods have local scope in that method only. Variables defined outside everything have global scope. Variables defined as members of a class have are accessible from any method within the class. We can also define variables as static, just like Java.

However, we can use static variables inside a method and they behave like global variables:

```
void foo()
{
        static int x = 0;
        x++;
        cout << x << endl;
}
```

If foo is called twice, on the second time, x is set to 2.

At this point it may be helpful to see a small example program. Here is a class that stores fractions and performs some small manipulations on them:

File: fraction.h

```
class Fraction {
        public:
                Fraction();
                void Set(int num, int denom);
                void Print();
                void Get(int &num, int &denom);
        private:
                int m_numerator, m_denominator;
```

```
};                  // Don't forget the ;
```

File: fraction.cpp

```
#include <iostream>
#include "fraction.h"
using namespace std;

Fraction::Fraction() {
        // Initial values
        m_numerator=0;
        m_denominator=1;
}

void Fraction::Set(int num, int denom) {
        m_numerator = num;
        m_denominator = denom;
}

void Fraction::Print() {
        cout << m_numerator << "/" << m_denominator << endl;
}

void Fraction::Get(int &num, int &denom) {
        num = m_numerator;
        denom = m_denominator;
}
```

File main.cpp

```
#include <iostream>
#include "fraction.h"
using namespace std;

int main()
{
      Fraction f1;      // Constructor invoked upon creation
      int a=0,b=0;

      f1.Print(); // Outputs   0/1
      f1.Set(1,2);
      f1.Print(); // Outputs  1/2
      f1.Get(a,b);
      cout << a << " " << b << endl;      // Outputs 1 2
      return 0;
}
```

C++ lets us use the keyword const to make things constant and unchangeable, but it can be used in many different ways that can be confusing.

```
const int x = 3;          and
int const x = 3;
```

both make x a constant that is an integer set to 3. You can't change x. In general, the thing to the left is what the const applies to, unless there is nothing there, in which case the const applies to the thing to the right.

Const can also be used for a return type. Consider this:

```
int& foo()
{
        static int x = 0;
        return x;
}

main:
        cout << foo()++ << endl;
        cout << foo() << endl;
```

This outputs 0 and then 1, because the ++ changes the return value which references the static variable. If you want to return a reference for efficiency purposes but don't want it changeable then you can make it const:

```
const int& foo()
```

Const can also be used with parameter passing. You might want to pass something by reference to save memory, but really don't want the parameter to be changed. In this case you can make it const:

```
void foo(const big_class &parameter)
```

Finally, const can be used in OOP to specify that a function can't change any instance (member) variables in the object. To do this stick const at the end of the function in the header and implementation file:

```
class Fraction {
                public:
                        void Print() const;
…
void Fraction::Print() const
{
        cout << m_numerator << "/" << m_denominator << endl;
}
```

You can combine these to be extra confusing:

```
const int& myfunction(const MyClass& parm) const
```

One nice thing that C++ allows that is not allowed in Java is we can overload operators. For example, let's say we would like to define the *= operator so it multiplies the first fraction by the second fraction, i.e. consider the following:

```
int main()
{
        Fraction f1,f2;
        f1.Set(1,2);
        f2.Set(3,4);
        f1*=f2;                         // f1 = 1/2*3/4  = 3/8
        f1.Print();                     // Outputs  3/8
        return 0;
}
```

We can accomplish by adding the following as a public method of fraction.h :

```
Fraction& operator*=(const Fraction &rtSide);
```

Then in fraction.cpp we add:

```
Fraction& Fraction::operator*=(const Fraction &rtSide)
{
      m_numerator *= rtSide.m_numerator;
      m_denominator *= rtSide.m_denominator;
      return *this;
}
```

This produces the desired result. The most typical use of overloaded operators is to overload the assignment operator, =, to perform more complicated tasks when we are setting two objects equal to one another (e.g., we might want to copy various data variables from the source to the target).

Arrays: Arrays are used much like in Java, but are defined a bit differently. The format for defining an array in C++ is:

```
type varname[SIZE];
```

If defined as a local variable, this will allocate enough space on the stack to store the entire array (i.e. SIZE*sizeof(type)). varname then holds the address of the first element in the array. Note that this is different from Java, which requires us to use the new operator which allocates the array data from the heap.

Accessing the contents of an array is just like in Java, e.g. varname[0] accesses the element at position 0, up to varname[SIZE-1].

Multi-dimensional arrays can be created by adding more brackets. For example, the following creates a 3-d array:

```
int three_d_array[10][10][10];
```

Passing an array as a parameter to a method requires some new syntax. To define an array parameter, use the definition of the array without any size, e.g.:

```
void someFunction(int arr[])
{
        arr[0]=4;
}
```

This would be invoked as:

```
int arr[100];
someFunction(arr);
```

Arrays behave as they are passed by reference, so upon return arr[0] will equal 4 in the caller's scope. Note that inside the method we lose the notion of how large the array is, so we must either know as the programmer or we should pass in another parameter that is the size of the array.

If you pass multidimensional arrays to a function you have to specify the size of everything but the topmost dimension; e.g. void someFunction(int arr[][100]);

If we define arrays as shown here, then the array is allocated on the stack. We can also declare a dynamic array which allocates the array from the heap (shown later).

Pointers

Manipulation of pointers is where C++ strays from Java. When we covered how variables are passed by reference, we alluded to the notion of pointers – the memory address where data is stored is actually passed along, resulting in a change to the original data. There is a method in C++ to explicitly get the address of an identifier. The addresses are called **pointers** and pointers are one of the primitive data types.

To declare something to be a pointer, use a * in the definition of the variable.
For example, the following defines a variable to be a pointer to a long:

```
long lngVal=0,  *lngPtr;
```

This declares two variables. One (lngVal) is a normal variable with enough space allocated to it to hold a long, initialized in this case to zero. The other (lngPtr) is a variable with enough space allocated to it to hold a memory address. It is intended that this memory address be that of a long somewhere in memory, although since pointers to different types could be mixed since all pointers are the same size, regardless of what they point to..

It is common to initialize pointers to NULL. This stores the value 0 into a pointer and can be used to test if a pointer has been set or not:

```
long *lngPtr = NULL;
```

Pointer Operators

The **&** or **address** operator returns the address of its operand. To use it, precede an identifier by & to retrieve its address. **(This is different from the & used in passing variables by reference!)** For example:

```
long lngVal=0,  *lngPtr;

cout << &lngVal  << endl;
```

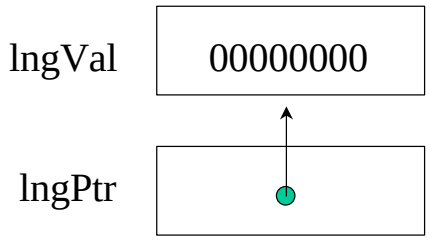
This will print out the address of the variable lngVal.

We can assign a memory address to a pointer variable. To assign the address of lngVal to lngPtr we use the assignment operator:

```
lngPtr = &lngVal;
```

This copies the address of lngVal into the contents of lngPtr:

Often this is depicted graphically:



What happens if we print the value of lngPtr? Let's look at two properties of lngPtr:

```
cout << lngPtr << endl;
cout << &lngPtr << endl;
```

The first statement prints out the contents of lngPtr. This would output the memory address of lngVal, which will appear as some hex number. The second statement prints out the address of lngPtr. Yes, we can get the address of a pointer just like any other variable! .

If we want to access the value being referenced by the pointer, then we must use the ***** or **indirection / dereferencing operator**. To use it, add the operator in front of the pointer to dereference:

```
cout << *lngPtr << endl;
```

This will output 0. The dereference operator follows the pointer and references the value being pointed at. In other words, this operator fetches the memory address stored in the pointer. Then it goes to that memory address and fetches the data stored at that address.

Sample exercise: What is the output of the following code?

```
int x, *xPtr=NULL, y, *yPtr=NULL;
x=10;
y=20;
xPtr = &x;
yPtr = &y;
cout << x << *xPtr << y << *yPtr << endl;
yPtr = xPtr;
cout << *yPtr << endl;
cout << xPtr << &x << endl;
```

The first cout statement will print the values of x and y:

```
10 10  20 20
```

The second cout statement will print 10 because yPtr is changed to the address of x:

```
10
```

The last statement will print the address of x, which will vary when run:

```
1045123 1045123
```

What is the output of the following?

```
int x, *xPtr=NULL, y, *yPtr=NULL;
x=10;
xPtr = &x;
*xPtr = 20;
cout << x << *xPtr << endl;
```

By setting xPtr to the address of x, and then changing it by dereference, the contents of x are changed and the output is 20:

```
20 20
```

What is wrong with the following?

```
int x, *xPtr=NULL, y, *yPtr=NULL;
x = 10;
*xPtr = 20;
cout << x << *xPtr << endl;
```

In this case, we are trying to store the value 20 into memory address NULL (0). This is not allowed and will likely cause the program to crash. We should only dereference a pointer after it is pointing to some allocated memory. In this case, the pointer is not pointing to any allocated memory. We can point to allocated memory by setting a pointer to another variable using the & operator. We can also use the **new** operator described next to allocate memory for us.

The above is a very common bug, so watch out and make sure that your pointers hold valid references!

Dynamic Variables

In the previous discussion, we created a variable and stored its address in a pointer variable. By now, you should be asking, but why? Why would you go to this much trouble if it is only an alternative way to do something you can already do? The ability to store and manipulate the addresses of variables allows us to create **dynamic** variables allocated from the heap.
Dynamic variables are created and destroyed as needed by the program itself. **new** is an operator that allocates a dynamic variable; **delete** is an operator that deallocates it. The format is :

```
ptrVar = new <type>
delete ptrVar;
```

Here is a code sample:

```
char*  charPtr=NULL;
MyClass*   classPtr=NULL;
char*  aCString=NULL;

charPtr = new char;                 // Allocate a character
classPtr  = new MyClass();          // Allocate a class (similar
to Java)
aCString = new char[10];            // Allocate 10 characters
```

```
*charPtr = 'A';
                          // Suppose MyClass has "intMemberVar"
(*classPtr).intMemberVar  = 55;
strcpy(aCString, "foo");

cout << aCString ;
cout << " "  << *charPtr  << "." ;
cout << (*classPtr.intMemberVar)  << endl;

delete  charPtr;         // Deletes necessary to avoid garbage
delete  intMemberVar;
delete[]  aCString;      // use [] to indicate deleting an array
```

This code segment produces the output:

foo A 55

In this example, the variables created by **new** to hold the data exist only from the execution of **new** to the execution of **delete**. The data is allocated from a large block of free memory called the **heap**. The locations that were allocated for them can now be allocated for some other variables. Here, we are talking about three variables, so space is not important. However, when you graduate to writing very large programs, this technique of using space only during the time that you need it becomes important.

The variables generated by new are called dynamic variables because they are created at run time. Dynamic variables are used just like any other variables of the same type, but pointer variables should only be assigned to one another, tested for equality, and dereferenced.

Important:

- Never try to store data into a pointer that does not have space allocated for it.

  int *xPtr;
  *xPtr = 10; ← NO! Memory has not been allocated for the data

- Never delete a pointer that was not allocated with new. If the pointer was assigned to an auto variable, there is no need to delete it.

  int *xPtr, x=5;
  xPtr = &x;
  delete xPtr; ← NO!

- Always delete a pointer that is allocated with new when you are finished! If you do not explicitly delete the memory, it will create garbage and waste memory.

  int *xPtr;
  xPtr = new int;

```
*xPtr = 5;
xPtr = NULL;          ← Lose handle to allocated memory!
```

In this case we changed xPtr, it is no longer pointing to the allocated memory! By changing the pointer, we lost the "handle" to the allocated memory.

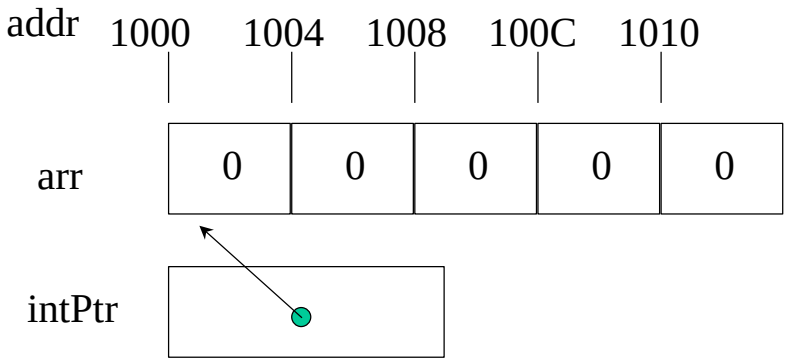
Pointer Arithmetic

A limited set of arithmetic can be performed on pointers. For example, we can add a value to a pointer. But what does it mean to increment a pointer? The answer is that the pointer is changed by an amount equal to the size of the type that the pointer is referencing.

For example, let's say that we declare an array of integers:

```
int arr[5]={0};
int *intPtr;

intPtr = &arr[0];          (equivalent to   intPtr = arr);
```

The allocation in memory looks something like this:



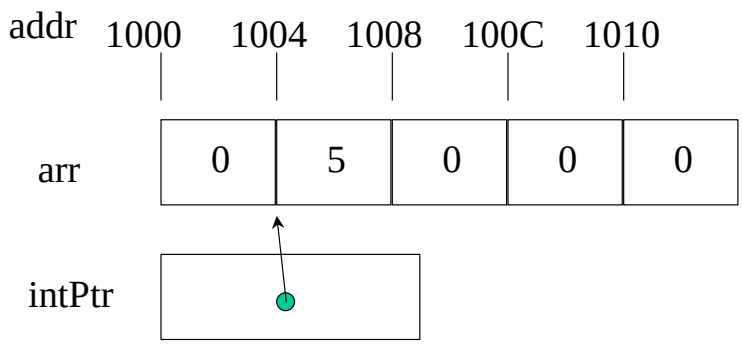
In this picture, intPtr is pointing to the beginning of the array, or it contains address 1000 (hex).

What happens if we do intPtr = intPtr + 1?

In normal arithmetic, we would get 1000 +1 or 1001. However, this is not the case with pointer arithmetic. Instead, the pointer is changed by the integer times the size of the object to which the pointer refers. In this case, each integer is 4 bytes. So by adding one to intPtr, we are actually adding 4, to hold 1004 and point to the next element in the array. If we execute the following:

```
intPtr = intPtr +1;
*intPtr = 5;
```

The picture is changed to:

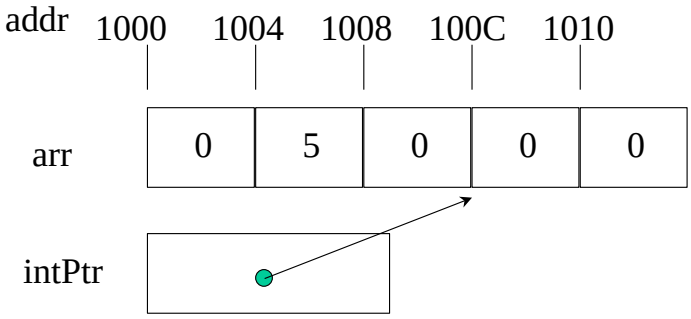


If we were to print out arr[1] then we would be outputting the value 5!

Similarly, if we now set:

```
intPtr +=2;
```

The pointer is changed to move down two elements, referencing 100C:



Exercise: What does the following code do?

```
char s[6]=”krunk”;              // Stores \0 at the end into s[5]
char *ptr1,*ptr2, c;

ptr1=&s[0];
ptr2=&s[4];
while (ptr1<ptr2) {
        c=*ptr1;
        *ptr1=*ptr2;
        *ptr2=c;
        ptr1++;
        ptr2--;
}
cout << s << endl;
```

Arrays and Pointers

Arrays and pointers are intimately related in C++. They may be used almost interchangeably. An array name is basically a constant pointer; it cannot be changed.

Consider the following pointer and array:

```
int arr[10], *intPtr=NULL;
intPtr = &arr[0];
```

The above is equivalent to:

```
intPtr = arr;
```

Array element arr[3] could also be referenced as:

```
*(intPtr + 3)
```

The number three above is the offset into the pointer. Since pointer arithmetic will add the proper amount based on the size of each element, we'll be guaranteed to get the fourth value (i.e. the value in arr[3]).

If we wanted the address of arr[3] we could use:

```
&arr[3]
```

this is equivalent to:

```
(intPtr + 3);
```

So we can use a pointer to access data within an array by just adding in the proper offset. We can also treat an array like a pointer!

```
&(arr + 3)
```

Returns the same thing as (intPtr + 3), because intPtr and arr are just both pointers to the beginning of the array.

We can even treat pointers like they are arrays:

```
intPtr[3]   will access the same element as arr[3]
```

To recap, we can apply almost all pointer operations to arrays, and vice versa:

```
int arr[10], intPtr=NULL;
```

```
intPtr = &arr[0];

*(intPtr + 3)   ==   arr[3];
*(arr + 3)      ==   intPtr[3];
(intPtr + 3)    ==   &arr[3];
(arr + 3)       ==   &intPtr[3];
```

One place where arrays are NOT like pointers is that you cannot change an array. Arrays are like constant pointers:

```
intPtr = intPtr + 1;        // VALID, intPtr now points to arr[1]
arr = arr + 1;              // INVALID, can't change an array!
```

Structures and Pointers to Structures (Linked Lists)

As we have seen from the programming languages textbook, structures and classes allow us to group together disparate data types into a single object. In this section we will look at the **struct** although everything we cover here also applies to a class (which you should hopefully be familiar with from learning Java).

To define a struct, use the keyword struct followed by the name of the structure. Then use curly braces followed by variable types and names:

```
struct StructName
{
        type1   var1;
        type2   var 2;
        …
        type3   var N;
};
```

Note the need for a semicolon at the end of the right curly brace!

The above defines a structure named "StructName". You can use StructName like it is a new data type. For example, the following will declare a variable to be of type "StructName":

```
StructName myVar;
```

To access the members (variables) within the structure, use the variable name followed by a dot and then the variable within the struct. This is called the member selector:

```
myVar.var1;
```

Here is an example structure:

```
struct  Recording
{
   string    title;                    // STL string
   string    artist;
   float     cost;
   int       quantity;
};
Recording  song;                       // Declaration of variable
```

The only aggregate operation defined on structures is assignment, but structures may be passed as parameters and they may be the return value type of a function. Assignment copies each member variable from the source to the destination struct. For example, the following is valid:

```
Recording song1,song2;

song1.title = "YMCA";
song1.artist = "Village People";
song1.cost = 10.50;
song1.quantity = 2;
song2 = song1;
cout << song2.title << endl;
```

This will print out "YMCA" as the contents of song1 get copied to song2. However, the default for assignment is to only copy one level deep, not a deep copy. For example, if one of the member variables referenced a linked list, we don't get a whole new copy of the linked list, we instead get a reference to the original linked list.

We can do other things like make an array of structs:

```
Recording songs[100];
songs[0].title = "YMCA";
songs[0].artist = "Village People";
… etc...
```

Linked Structures

Dynamic variables combined with structures can be linked together to form dynamic lists. We define a structure (called a node) that has at least two members: next (a pointer to the next node in the list) and component (the type of the items on the list). For example, let's assume that our list is a list of integers.

```
struct NodeType
{
   int num;                    // Some numeric value for this node
   NodeType  *next;            // Pointer to a NodeType
};

NodeType  *headPtr;                 // Pointer to the first thing in the list
NodeType  *newNodePtr;              // extra pointer
```

To form dynamic lists, we link variables of NodeType together to form a chain using the next member. We get the first dynamic list node and save its address in the variable headPtr. This pointer is to the first node in the list. We then get another node and have it accessible via newNodePtr:

```
headPtr = new NodeType;
newNodePtr = new NodeType;
```

Next, let's store some data into the node pointers. To access the structure, we have to first de-reference the pointer (using *) and then we need to use the dot notation to access the member of the structure:

```
(*headPtr).num = 51;
(*headPtr).next = NULL;
```

Instead of using the * and the . separately, C++ supports a special operator to do both simultaneously. This operator is the arrow: -> and it is identical to dereferencing a pointer and then accessing a structure:

```
newNodePtr->num = 55;
newNodePtr->next = NULL;
```

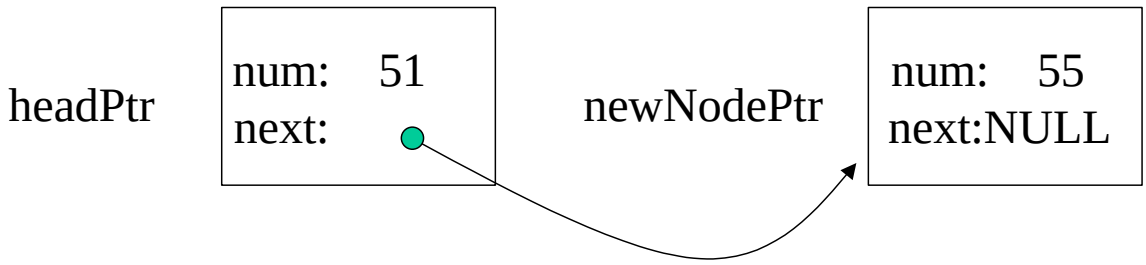
is identical to

```
(*newNodePtr).num = 55;
(*newNodePtr).next = NULL;
```

Right now we have two separate NodeTypes. We can link them together to form a linked list by having the next field of one pointing the address of the next node:

```
headPtr->next = newNodePtr;
```

We now have a picture that looks like:

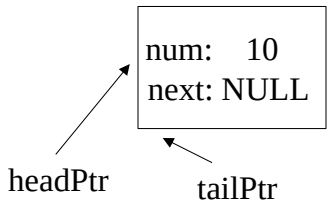


We just built a linked list consisting of two elements! The end of the list is signified by the *next* field holding NULL.

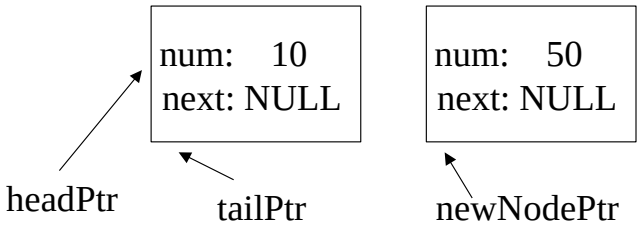
We can get a third node and store its address in the next member of the second node. This process continues until the list is complete. The following code fragment reads and stores integer numbers into a list until the input is –1:

```
struct NodeType
{
 int num;
 NodeType *next;
};

int main()
{
 NodeType *headPtr, *newNodePtr, *tailPtr, *tempPtr;
 int temp;

 headPtr = new NodeType;
 headPtr->next = NULL;
 tailPtr = headPtr;                  // Points to the end of the list
 cout << "Enter value for first node" << endl;
 cin >> headPtr->num;           // Require at least one value

 cout << "Enter values for remaining nodes, with -1 to stop." << endl;
 cin >> temp;
 while (temp!=-1) {
       // First fill in the new node
       newNodePtr = new NodeType;
       newNodePtr->num = temp;
       newNodePtr->next = NULL;
       // Now link it to the end of the list
       tailPtr->next=newNodePtr;
       // Set tail to the new tail
       tailPtr = newNodePtr;
       // Get next value
       cin >> temp;
 }
```

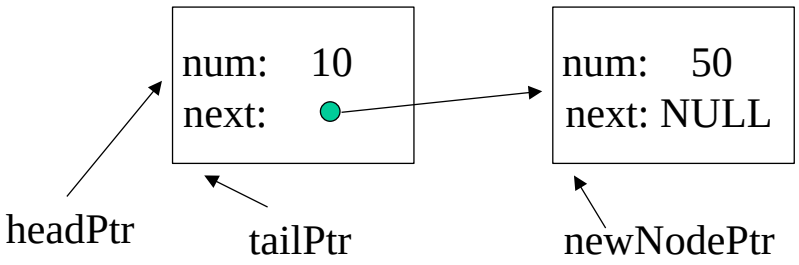
This program (it is incomplete, we'll finish it below) first allocates memory for headPtr and inputs a value into it. It then sets tailPtr equal to headPtr. tailPtr will be used to track the end of the list while headPtr will be used to track the beginning of the list. For example, let's say that initially we enter the value 10:



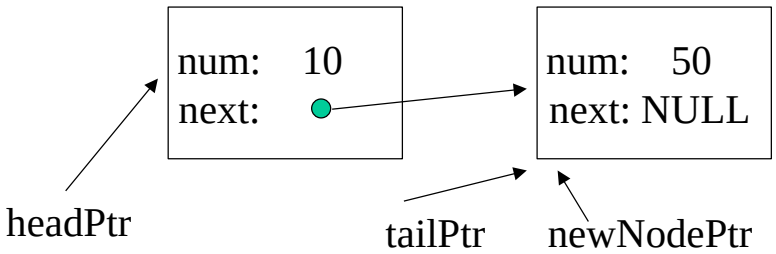
Upon entering the loop, let's say that we enter 50 which is stored into temp. First we create a new node, pointed to by newNodePtr, and store data into it:



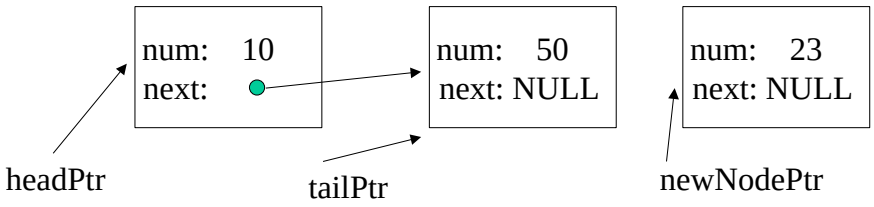
Then we link tailPtr->next to newNodePtr:



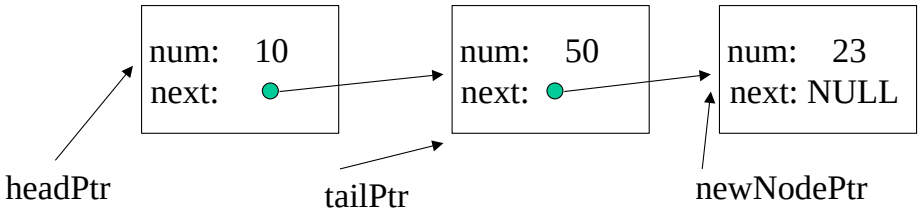
Finally we update tailPtr to point to newNodePtr since this has become the new end of the list:



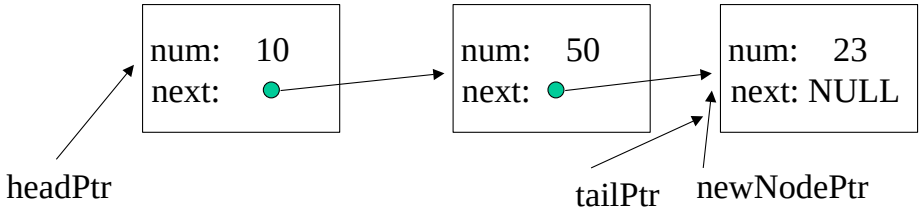
Let's say that the next number we enter is 23. We will repeat the process, first allocated a new node pointed to by newNodePtr, and filling in its values:



Then we link tailPtr to newNodePtr:



Finally we update tailPtr to point to the new end of the list, newNodePtr:



The process shown above continues until the user enters –1. Note that this allows us to enter an arbitrary number of elements, up until we run out of memory! This overcomes limitations with arrays where we need to pre-allocate a certain amount of memory (that may turn out later to be too small).

Lists of dynamic variables are traversed (nodes accessed one by one) by beginning with the first node and accessing each node until the next member of a node is NULL. The following code fragment prints out the values in the list.

```
cout << "Printing out the list" << endl;
tempPtr = headPtr;
while (tempPtr!=NULL) {
      cout << tempPtr->num << endl;
      tempPtr=tempPtr->next;
}
```

tempPtr is initialized to headPtr, the first node. If tempPtr is NULL, the list is empty and the loop is not entered. If there is at least one node in the list, we enter the loop, print the member component of tempPtr, and update tempPtr to point to the next node in the list. tempPtr is NULL when the last number has been printed, and we exit the loop.

Once we have printed out the data, we're not done! Before we exit the program, we should make certain to free up the memory we allocated to prevent memory leaks. We can do so in a loop similar to the one we used to print out the list:

```
// Now free the dynamically allocated memory to prevent memory leaks
 while (headPtr!=NULL) {
       tempPtr=headPtr;
       headPtr=headPtr->next;
       delete tempPtr;
 }
}        // End program (piecing together all of the above)
```

This loop goes through and frees each node until we reach the end.

Note that we used two pointers above, tempPtr and headPtr. What is wrong with the following?

```
while (headPtr!=NULL) {
      delete headPtr;
      headPtr=headPtr->next;
}
```

Because the types of a list node can be any data type, we can create an infinite variety of lists. Pointers also can be used to create very complex data structures such as stacks, queues, and trees that are the subject of more advanced computer science courses.

Using Classes as Dynamic Data Structures

The previous code would work identically if we replace the struct with a class:

```
struct NodeType
{
 int num;
 NodeType *next;
};
```

to

```
class NodeType
{
        public:
                int num;
                NodeType *next;
        private:
};
```

We don't have to change anything else to the code. If we use a class instead of a struct though, we have the option of using all of the niceties of classes (inheritance, define methods, overloading, overriding, hiding data with private, etc.) You can actually do many of these things with structs as well, but normally a struct is only used to store data.

Other Things

There are many other aspects of C++ that are different than Java, but in our limited time we must stop here. A short list of some of the differences from the C++ perspective are listed below:

- Multiple inheritance
- Standard Template Library
- Goto
- Function Pointers
- Nested Classes
- Lack of built-in multithreading support
- Preprocessing directives