C++ Lecture 6
Pointers

CSci 588: Data Structures, Algorithms and Software Design

http://www.cplusplus.com/doc/tutorial/pointers/

Overview

Pointers

1. Address-of Operator
2. Dereference Operator
3. Declaring Pointers
4. Pointers and Arrays
5. Pointer Initialization
6. Pointer Arithmetic
7. Pointers and String Literals
8. Pointers to Pointers
9. void pointers
10. invalid pointers and null pointers
11. pointers to functions

Dynamic Memory

1. new and new []
2. delete and delete []
3. Dynamic Memory in C

Pointers

Pointers

Pointers are so named because they "point" to an area of memory.

In a C++ program, the memory of a computer is a succession of memory cells, each one byte in size, and each with a unique address.

For example, the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776.

Pointers

When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address).

Typically, C++ programs do not actively decide the exact memory addresses where its variables are stored. Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime.

However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

Address-of Operator (&)

Address-of Operator

The address of a variable can be obtained by using the Address-of operator (&) as follows:

```
address = &my_variable;
```

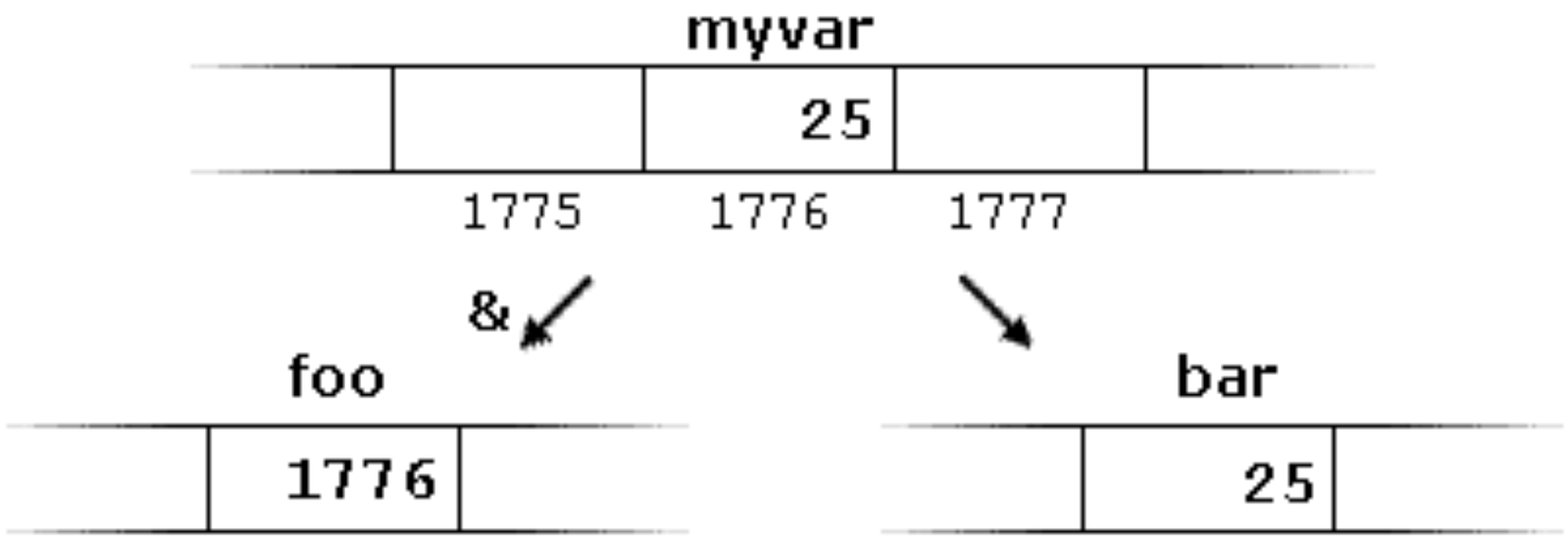
This assigns the address of 'my_variable' to the 'address' variable. When we deal with the address variable, we are referring to the address of my_variable, *not* its value.

Address-of Operator

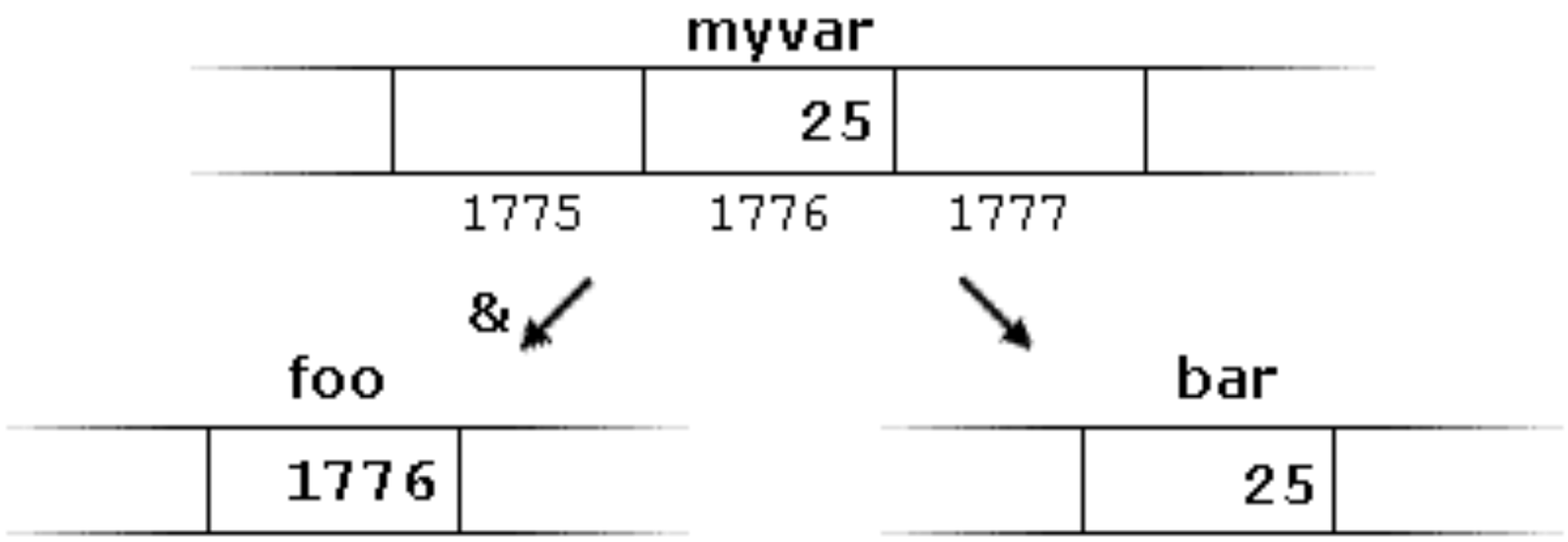
Consider the following:

```
myvar = 25;
foo = &myvar;
bar = myvar;
```

The values contained in each variable are:



Address-of Operator




```
myvar = 25;
foo = &myvar;
bar = myvar;
```

The variable 'myvar' has 25 assigned to it, and it is at address 1776. foo is assigned its address, so the value of foo is 1776 (at some other address) and the value of bar is 25, the same as myvar, but it is at some other address as well.

Address-of Operator

The variable that stores the address of another variable (like foo in the previous example) is what in C++ is called a pointer. Pointers are a very powerful feature of the language that has many uses in lower level programming.

Dereference Operator (*)

Dereference Operator

A variable that stores the address of another variable is called a *pointer*. Pointers are said to "point to' the variable whose address they store.

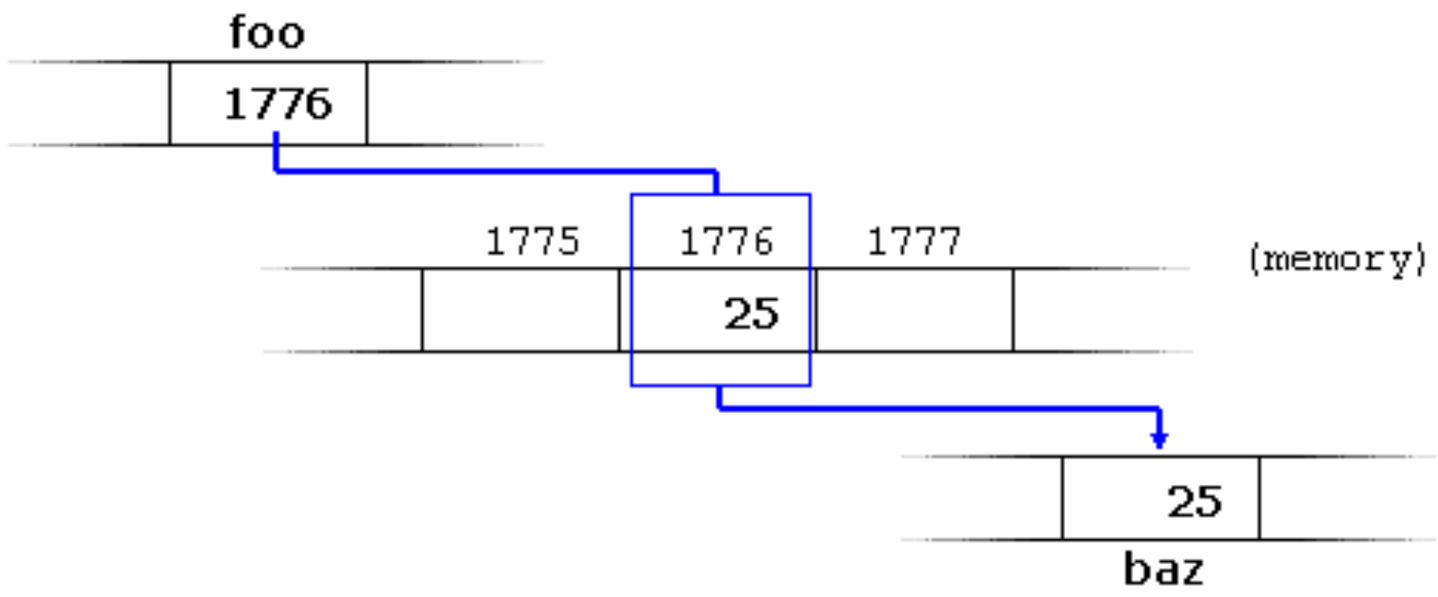
Pointers can also be used to directly access the variable they point to. This is done by preceding the pointer name with *, the *dereference operator*. The operator itself can be read as "the value pointed to by":

```
baz = *foo;
```

Dereference Operator

baz = *foo;

This could be read as :"baz is set equal to the value pointed to by foo" and the statement would assign the value 25 to baz, since foo is 1776, and the value pointed to by 1776 is 25 (given the previous example):



Dereference Operator

It is important to clearly differentiate that foo refers to value 1776, while *foo (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the dereference operator (I have added an explanatory comment of how each of these two expressions could be read):

```
baz = foo;   // baz equal to foo (1776)
baz = *foo;  // baz equal to value pointed to by foo (25)
```

Dereference Operator

The reference and dereference operators are thus *complementary*.

- & is the *address-of operator*, and can be read simply as "address of"
- * is the *dereference operator*, and can be read as "value pointed to by"

Generally, they work in opposite manners to each other, for example:

```
bar = 25;
otherbar = *(&bar); //this will set otherbar to 25
```

Dereference Operator

Given:

```
myvar = 25;
foo = &myvar;
```

Right after these two statements, the following are all true:

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

Dereference Operator

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

The first expression is quite clear, considering that the assignment operation performed on myvar was myvar=25.

The second one uses the address-of operator (&), which returns the address of myvar, which we assumed it to have a value of 1776.

The third one is somewhat obvious, since the second expression was true and the assignment operation performed on foo was foo=&myvar.

The fourth expression uses the dereference operator (*) that can be read as "value pointed to by", and the value pointed to by foo is 25.

Dereference Operator

Also, the following expression will always be true:

*foo == myvar

As foo is the address of myvar, so the value pointed to by it will be the value of myvar.

Declaring Pointers

Declaring Pointers

Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float. Once dereferenced, the type of the variable needs to be known.

Because of this, the declaration of a pointer needs to include the data type the pointer is going to point to:

type * name

Declaring Pointers

type *name

Where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
int *number;
char *character;
double *decimals;
```

This can be read as *number is an int; *character is a char, *decimals is a double, and so on.

Declaring Pointers

```
int *number;
char *character;
double *decimals;
```

All three are declarations of pointers, but each one refers to a different data type. On the other hand, as they are pointers, they each are likely to occupy the same amount of space in memory (pointers are 64 bits on 64 bit machines, and 32 bits on 32 bit machines).

However, the data types they point to are not all the same size (chars are 1 byte, ints are 4 bytes, and doubles are 8 bytes typically).

Declaring Pointers

Note that similar to arrays and [], when the asterisk (*) is used to declare a pointer, this is different than using it as a deference operator.

That's why in general it's best to read pointer declarations as follows:

```
int *number;
char *character;
double *decimals;
```

Where this means *number is an int; *character is a char, *decimals is a double, and so on.

Declaring Pointers

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int *p1, *p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Declaring Pointers

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int *p1, *p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Note that in the example, this line is important:

```
int *p1, *p2;
```

If it was:

```
int* p1, p2;
```

Then p1 would be a pointer, and p2 would not be a pointer.

Pointers and Arrays

Pointers and Arrays

Arrays are closely related to pointers, in fact in general an array is just a pointer with some different syntax. It's always possible to convert between an array and a pointer of the proper type:

```
int myarray [20];
int * mypointer;
```

In this case, the following assignment is valid:

```
mypointer = myarray;
```

However the following is not, as an array always represents the same block of memory.

```
myarray = mypointer;
```

Pointers and Arrays

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

As shown above, pointers and arrays support the same operatons with the same meaning for both. The main difference is that pointers can be assigned new addresses while arrays cannot.

Pointers and Arrays

In the lecture about arrays, brackets ([]) were explained to specify the index of an element in an array. What they actually do is act as a dereferencing operator known as the *offset operator*.

They dereference the variable they follow just as the asterisk (*) does, but also add the number between the brackets to the address being dereferenced. For example:

```
a[5] = 0;        // a [offset of 5] = 0
*(a+5) = 0;      // pointed by (a+5) = 0
```

The two expressions are equivalent and valid, not only if *a* is a pointer, but also if *a* is an array. Remember that an array variable can also be used as a pointer to it's first element.

Pointer Initialization

Pointer Initialization

Pointers can be initialized to point to specific locations when they are defined:

```
int myvar;
int * myptr = &myvar;
```

The resulting state of variables after this code is the same as after:

```
int myvar;
int * myptr;
myptr = &myvar;
```

Pointer Initialization

When pointers are initialized, what is initialized is the address they point to (myptr), never the value they point to (*myptr), therefore don't confuse:

```
int myvar;
int * myptr = &myvar;
```

With:

```
int myvar;
int * myptr;
*myptr = &myvar;
```

The asterisk in the pointer declaration (2nd line) just indicates that the variable is a pointer, it is not a dereference operator (like line 3). Also note the spaces are not relevant, and don't change the meaning of an expression.

Pointer Initialization

Pointers can be initialized to either the address of a variable (line 2) or the value of another pointer (or array), as in line 3:

```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

Pointer Arithmetic

Pointer Arithmetic

Arithmetic operations on pointers are a bit different than on regular integer types.

First, only addition and subtraction are allowed; as other operations don't make sense for pointers (which just refer to memory addresses).

Also, addition and subtraction work differently, depending on the size of the data type to which the pointer points.

Pointer Arithmetic

Given:

```
char *mychar;
short *myshort;
long *mylong;

++mychar;
++myshort;
++mylong;
```

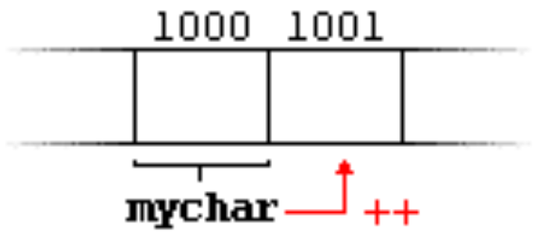
If these start at address 1000, 2000 and 3000, respectively, the their values will end up as 1001, 2002, and 3004 because of the size of the data the pointers point to.

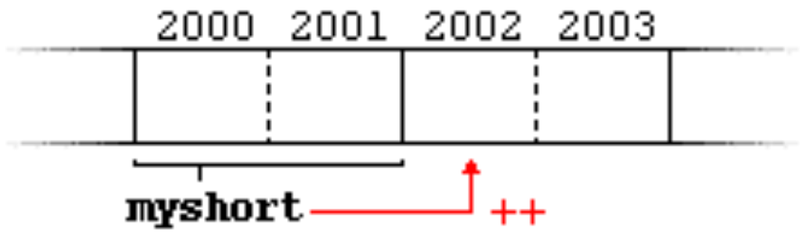
Pointer Arithmetic

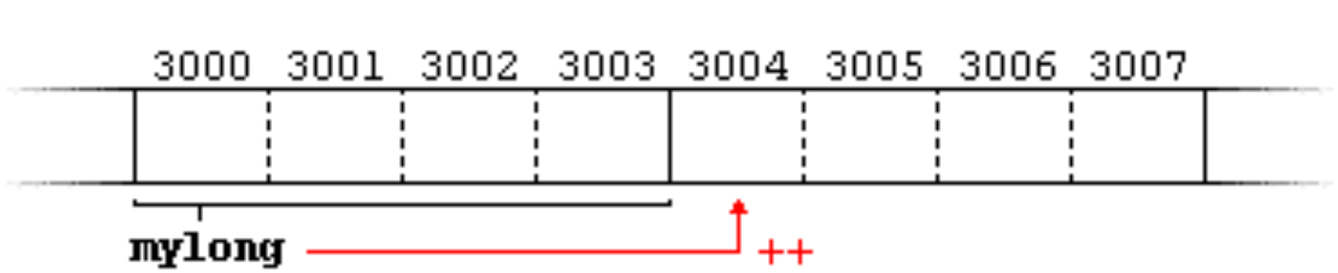
Given:

```
char *mychar;
short *myshort;
long *mylong;

++mychar;
++myshort;
++mylong;
```







If these start at address 1000, 2000 and 3000, respectively, the their values will end up as 1001, 2002, and 3004 because of the size of the data the pointers point to.

Pointer Arithmetic

This is applicable both when adding and subtracting any value. For example:

```
mychar = mychar + 1;
myshort = mychar + 1;
mylong = mychar + 1;
```

Would also end up as 1001, 2002, and 3004. Or:

```
mychar = mychar + 3;
myshort = mychar + 3;
mylong = mychar + 3;
```

Would end up as 1003, 2006 and 3012.

Pointer Arithmetic

It is important to be careful with the increment (++) and decrement (--) operators on pointers, especially in conjunction with the dereference operator (*). It's important to understand the precedence of operators (personally I just used parentheses to be clear, but not all programmers are good like that).

Pointer Arithmetic

This becomes complicated as ++ can be either a postfix or prefix operator, and * is a prefix operator as well. For example:

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

Pointer Arithmetic

A typical, but not so simple statement is:

```
*p++ = *q++;
```

This is roughly equivalent to:

```
*p = *q;
++p;
++q;
```

Because ++ has a higher precedence than *, both p and q are incremented, but because both increment operators are used as postfix and not prefix, the value assigned to *p is *p before they are both incremented. After the assignment, both are incremented. (This could be used, for example, to copy two arrays).

Again, I highly recommend using parenthesis to prevent confusion with these type of statements.

Pointers and Const

Pointers and Const

Pointers can be used to access a variable using it's address, and may also be used to modify the value at that address. It is also possible to declare pointers to values that can access the value pointed to and read it, but not modify it. This can be done using the const keyword.

For example:

```
int x;
int y = 10;
const int * p = &y;
x = *p;             // ok: reading p
*p = x;             // error: modifying p, which is const-qualified
```

Pointers and Const

Here p points to a variable, but it points to it in a const-qualified manner, meaning that it can read the value pointed but not modify it. Note also, that the expression &y is of type int*, but this is assigned to a pointer of type const int*. This is allowed: a pointer to non-const can be implicitly converted to a pointer to const. But not the other way around, as this would allow to get around the const qualifier.

```
int x;
int y = 10;
const int *p = &y;
x = *p;             // ok: reading p
*p = x;             // error: modifying p, which is const-qualified
```

Pointers and Const

```
// pointers as arguments:
#include <iostream>
using namespace std;

void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}

void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
    ++current;     // increment pointer
  }
}

int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

```
11
21
31
```

One great example of using a const pointer is for a function header (which I've talked about before with vectors). The idea is similar.

This allows you to pass a pointer to a function (which is quick) and not have to worry about that function modifying what's at that pointer.

Pointers and Const

```
// pointers as arguments:
#include <iostream>
using namespace std;

void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}

void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
    ++current;     // increment pointer
  }
}

int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

```
11
21
31
```

Note that print_all uses pointers that point to constant elements. These pointers can still be incremented or assigned different addresses, but they cannot modify the contents of what they point to.

Note that there is a problem with this code -- what is it?

Pointers and Const

This is where a second dimension of "const"-ness can be added to pointers. The pointers themselves can be const. This is specified by appending const to the pointed to type (after the asterisk:

```
int x;
      int *       p1 = &x;  // non-const pointer to non-const int
const int *       p2 = &x;  // non-const pointer to const int
      int * const p3 = &x;  // const pointer to non-const int
const int * const p4 = &x;  // const pointer to const int
```

Pointers and Const

The syntax with const and pointers is somewhat confusing and determining when to use which type requires some experience and comfort with c++ programming. However, it's good to get them right, as using const appropriately can significantly improve your code safety and correctness.

Pointers and Const

To add a little more confusion, const can come before or after the type, so the two following statements are equivalent:

```
const int * p2a = &x;  //      non-const pointer to const int
int const * p2b = &x;  // also non-const pointer to const int
```

As with the spaces surrounding asterisks, where the const goes is simply a matter of style and does not change what the program does at all.

Pointers and String Literals

Pointers and String Literals

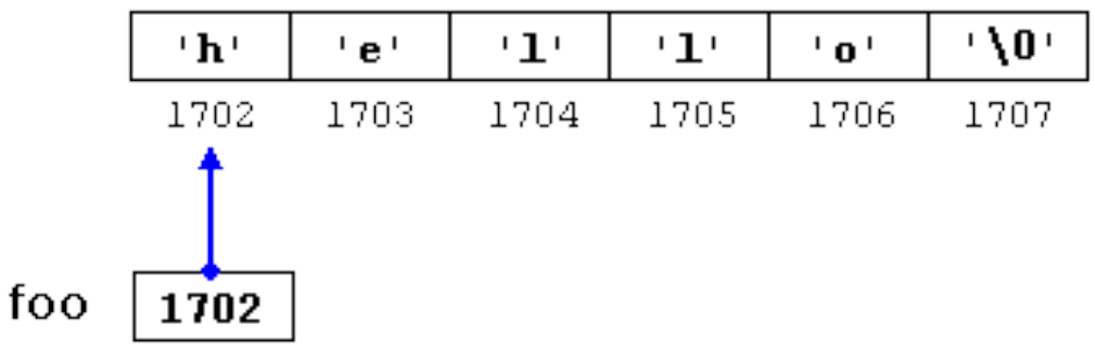
String literals (as opposed to std::string class) are arrays containing null terminated character sequences. We've used them a bit in class, to be directly sent to cout, to initialize strings, and to initialize arrays of characters.

Pointers and String Literals

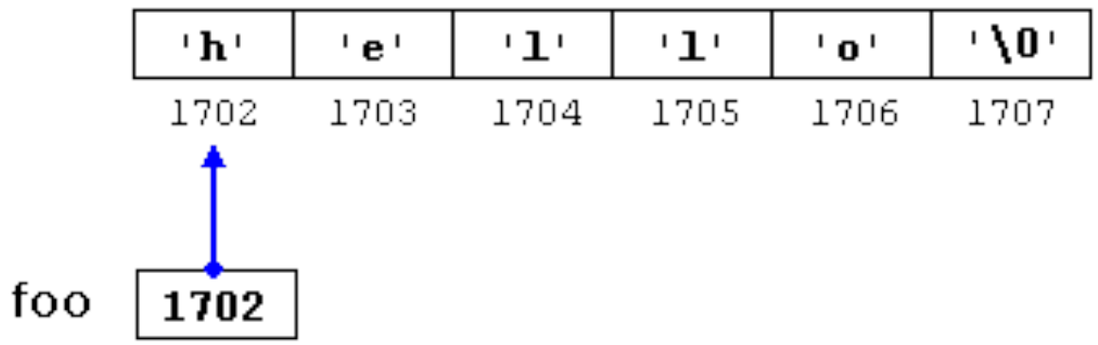
String literals can also be accessed directly. They are arrays of the proper array type to contain all its characters plus the null-terminating character, with each each of the elements being type const char (as literals they can never be modified). For example:

```
const char * foo = "hello";
```

This declares an array with the literal representation for "hello", and then a pointer to the first element called foo. If this is stored starting at memory address 1702, it can be represented as:



Pointers and String Literals



Note that \0 is a NULL character. Also, foo is a pointer and contains the value 1702 (the address of the first element in the array), not 'h' or 'hello'.

So the pointer foo points to a sequence of characters, and because pointers and arrays are handled essentially the same, foo can be used to access the characters within this array in teh same way, with both of the following have the value 'o':

```
*(foo+4)
foo[4]
```

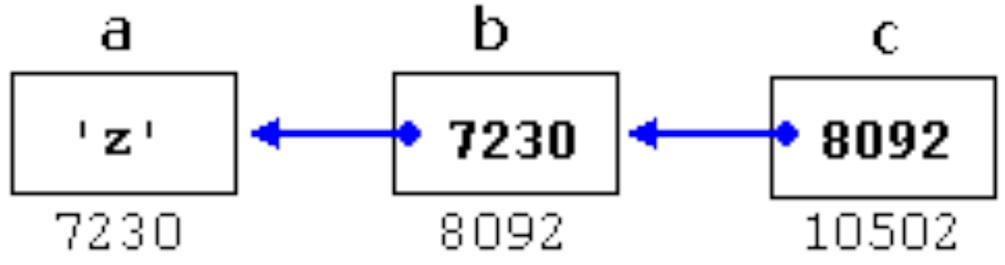
Pointers to Pointers

Pointers to Pointers

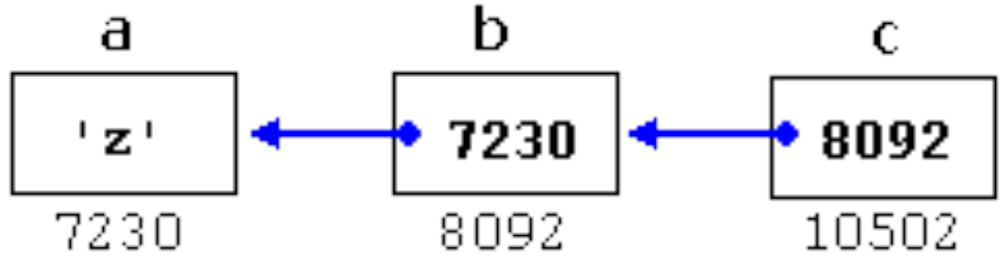
C++ also allows the use of pointers that point to other pointers, which in turn point to data (or even other pointers). The syntax simply requires an asterisk (*) for each level of indirection in the declaration of the pointer:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

This, assuming randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:



Pointers to Pointers



With the value of each variable represented inside it's corresponding cell, and their respective memory addresses represented by the value under them.

The new thing here is that the variable c, which is a pointer to a pointer, can be used in three different levels of indirection, each with a different value:

- c is of type char** and a value of 8092
- *c is of type char* and a value of 7230
- **c is of type char and a value of 'z'

Void Pointers

Void Pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of a type. Therefore void pointers are pointers that point to a value that has no type (or is of an unknown type), and thus also an undetermined length and undetermined dereferencing properties.

Void Pointers

This gives void pointers lots of flexibility, as they can point to any data type, from an int, to a float, to a char, to a long, etc. However, they are limited in that the data pointed to by them can not be directly dereferenced (which makes sense as the compiler would need to know the type of what's being dereferenced).

Therefore, before dereferencing a void pointer, it first needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

Void Pointers

One possible use for a void pointer is to pass generic parameters to a function:

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

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

```
y, 1603
```

Void Pointers

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

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

```
y, 1603
```

Remember that sizeof is an operator/function in C/C++ that returns the size (in bytes) of it's argument. For non dynamic data types the value is always constant. So for example, sizeof(char) is always 1 because a char is 1 byte.

Invalid pointers and NULL pointers

Invalid Pointers and Null Pointers

In principle, pointers are meant to point to valid addresses, such as the address of a variable or the address of an element in an array. However, pointers can also point to any address, including adresses that do not refer to any valid element (as we've seen more than a few times). Some typical examples of this are *uninitialized pointers* and pointers to non-existent elements of an array:

```
int * p;                  // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

Here, neither p nor q point to an address known to contain a value, but none of the above statements will cause an error.

Invalid Pointers and Null Pointers

In C++, pointers are allowed to take any address value, no matter if there is actually something at that address or not. As we've seen in these cases, dereferencing a pointer like this can cause an error.

But in general, accessing such a pointer causes undefined behavior, ranging from an error during runtime (like a seg fault) or accessing some random value.

Invalid Pointers and Null Pointers

Sometimes however, a pointer does need to point to nowhere, and not just an invalid address. In this case, there is a special value (with a few different representations) that any pointer can take: a *null pointer value*. This can be expressed as:

```
int * p = 0;
int * q = nullptr;
int * r = NULL;
```

With all of the above comparing equal to each other, as nullptr and NULL are just constants defined as 0.

Invalid Pointers and Null Pointers

It is important to not confuse NULL pointers with void pointers. A null pointer is a *value* that any pointer can take to represent that it is pointing to nowhere or nothing. A void pointer is a type of pointer that can point to somewhere without having a specific type.

A null pointer refers to the *value* of the pointer, while a void pointer refers to the *type* of the data it points to (or lack thereof).

Pointers to Functions

Pointers to Functions

C++ allows pointers to functions. The typical use for this is passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except the name of the function is enclosed between parentheses () and an asterisk (*) is placed before the name:

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  m = operation (7, 5, subtraction);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

```
8
```

Pointers to Functions

In the previous example, minus is a pointer to a function that has two parameters of type int. It is directly initialized to point to the function *subtraction*:

```
int (* minus)(int,int) = subtraction;
```

Dynamic Memory

Dynamic Memory

Up to now (apart from vectors), all the memory needs of your programs were determined before the program executed by having static sizes for arrays.

Unfortunately, this isn't particularly useful - but fortunately, C++ provides dynamic memory allocation with the operators *new* and *delete*.

Operators new and new[]

Operators new and new[]

Dynamic memory is allocated using the *new* operator. *new* is followed by a data type specifier and, if a sequence of more than one element is required, the number of elements is within brackets []. It returns a pointer to the beginning of the new block of memory allocated.

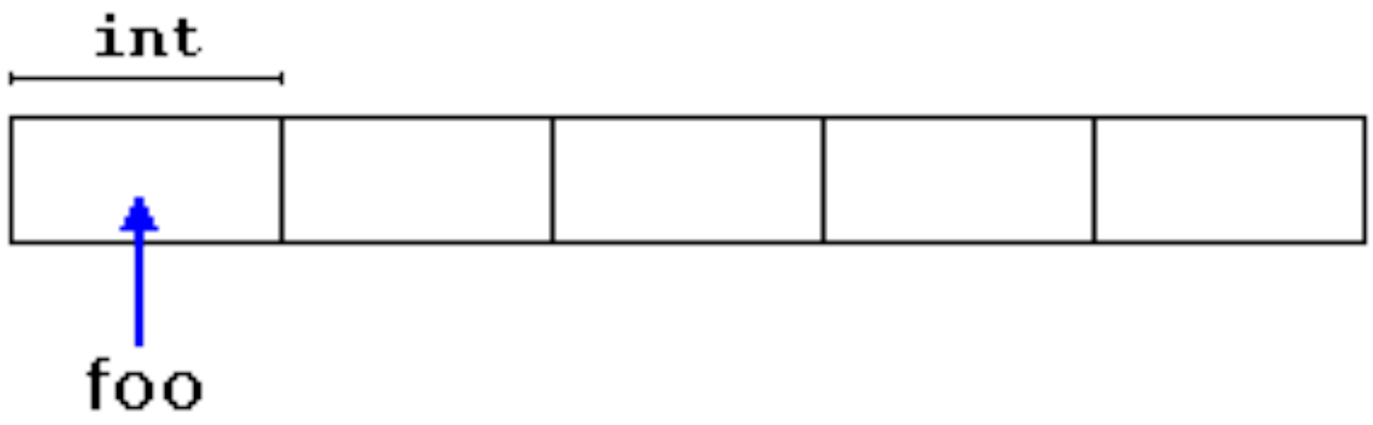
```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory for a single element of the type *type*; and the second is used to allocate a block (an array) of elements of type *type*, where *number_of_elements* is an integer value representing the amount:

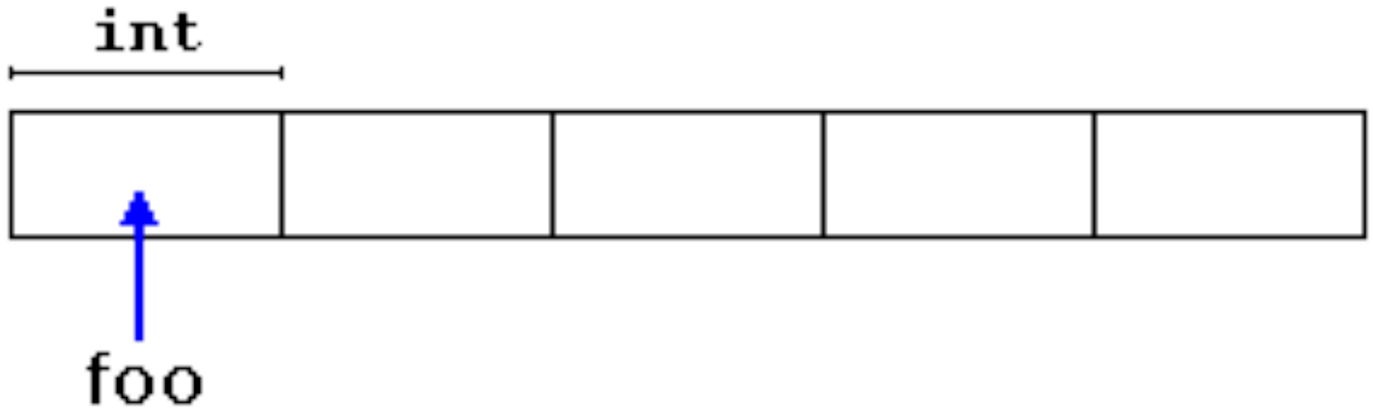
```
int * foo;
foo = new int [5];
```

Operators new and new[]

Using this operator, the system dynamically allocates space for five elements of type *int* and returns a pointer to the first element of the sequence, which is assigned to the pointer foo. After this statement, foo points to a block of memory with space for five elements of type int:



Operators new and new[]



Here, foo is a pointer, and therefore the first element can be addressed with foo[0] or *foo, the second element can be addressed with foo[1] or *(foo + 1), and so on - just like any other pointer.

The biggest difference between an array initialized with new vs. a statically initialized array is that the value within the [] and the new operator *does not* need to be a constant value, so you can allocate dynamic amounts of memory at runtime.

Operators new and new[]

Here, foo is a pointer, and therefore the first element can be addressed with foo[0] or *foo, the second element can be addressed with foo[1] or *(foo + 1), and so on - just like any other pointer.

The biggest difference between an array initialized with new vs. a statically initialized array is that the value within the [] and the new operator *does not* need to be a constant value, so you can allocate dynamic amounts of memory at runtime.

Operators new and new[]

When dynamic memory is requested with new (or malloc), it's allocated by the operating system from the memory heap. However, memory is a limited resource (your computer only has a limited amount of RAM), so there are no guarantees that all requests for memory are going to be granted by the system.

Operators new and new[]

Because of this, c++ has to mechanisms for checking to see if a memory allocation was successful.

The first, used by default, is to throw an exception, when a simple declaration is used:

```
foo = new int [5];  //if allocation fails, an exception is thrown
```

Operators new and new[]

The other is specified by using nothrow, a special object, as an argument to new:

```
foo = new (nothrow) int [5];
```

In this case, foo will be NULL (or nullptr) if the memory was not allocated:

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

Generally, using nothrow is less efficient than the default method, so it is rarely used. However we haven't gotten to exception handling yet. :)

Operators delete and delete[]

Operators delete and delete[]

Unlike other programing languages which have garbage collection (or memory management), when you allocate memory in C/C++, it is not automatically freed up when it is done being used. You need to do this yourself manually with the delete operator.

```
delete pointer;
delete[] pointer;
```

Here, the first statement releases the memory of a single element allocated using new, and the second releases the the memory allocated for the number of elements specified within the brackets ([]) when it was allocated by new.

Operators delete and delete[]

The value passed to delete can be either a pointer to a memory block previously assigned by new, or a null pointer. In the case of a null pointer, delete does nothing.

Operators delete and delete[]

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p = new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Note that the value used to determine how large the array p is, is specified by the user input, not a constant value.

Operators delete and delete[]

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p = new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Note that it is possible for a user to enter an i larger than the amount of memory available. In this case, the program will respond with an error message that that much memory could not be allocated.

Dynamic Memory in C

Dynamic Memory in C

While C++ uses the new and delete operators to handle dynamic memory (which syntactically are a lot easier to use), C used the more archaic malloc and free (along with calloc and realloc) functions:

```
int *c_array = (int*)malloc(sizeof(int) * 10);
int *cpp_array = new int[10];

free(c_array);
delete [] cpp_array;
```

Note that if you allocate an array with malloc, you can't release the memory with delete, and vice versa. Note that malloc always returns a void pointer, so it needs to be cast to the appropriate data type.