Introduction to C++

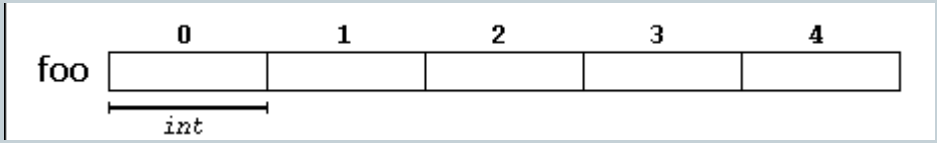
Outline

- Arrays
- Character Sequences
- Pointers

Arrays

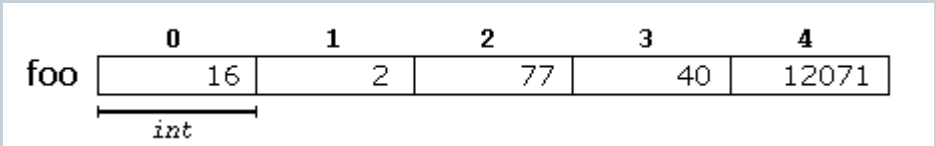
- Arrays in C++ are very similar to Java arrays
- To declare an array:

```
int foo [5];
```



- To declare and initialize:

```
int foo [5] = { 16, 2, 77, 40, 12071};
```

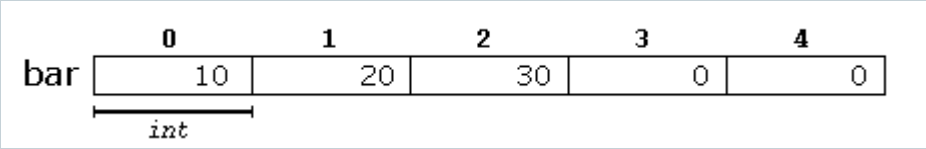



```
int foo [] = { 16, 2, 77, 40, 12071};
```

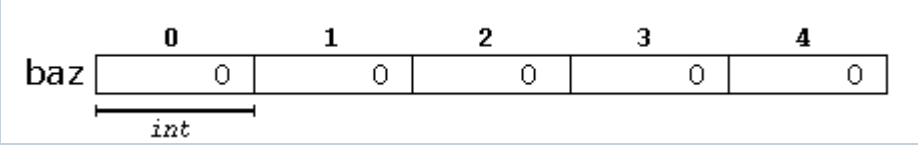
Arrays

- If you declare and array and initialize it with fewer values than specified, the remaining values will be the default

```
int bar [5] = { 10, 20, 30 };
```

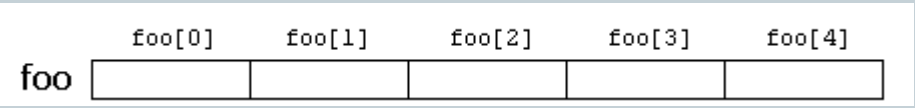



```
int baz [5] = { };
```



Accessing Array Values

- Just like in Java, provide the name[index]:



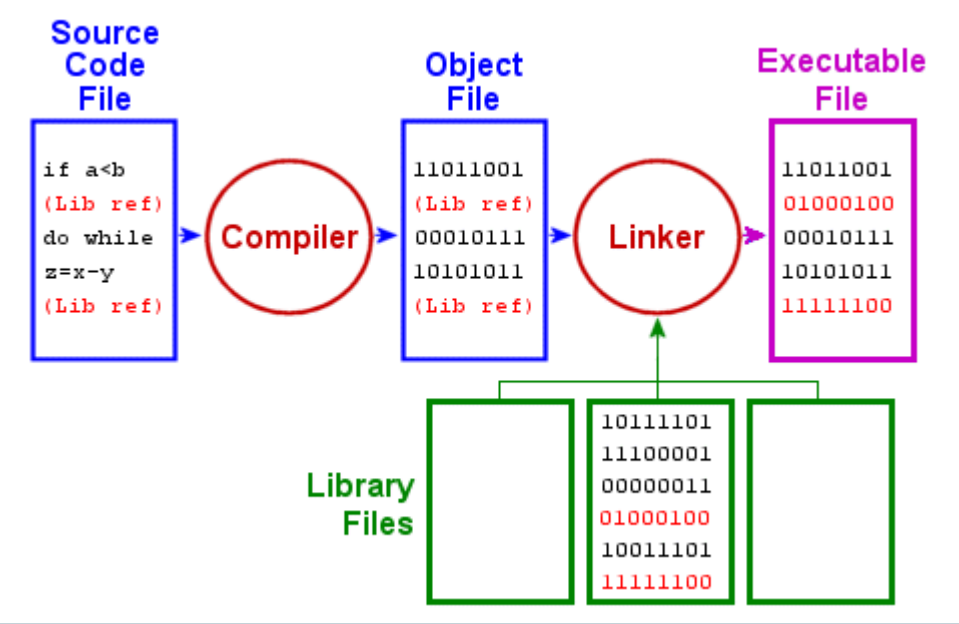

```
foo[2] = 75;
x = foo[2];
```

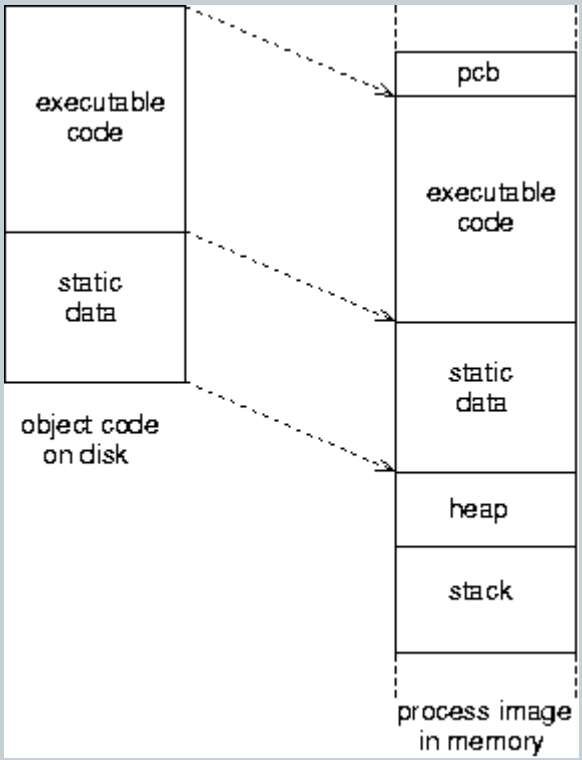
- What will happen in the following code?

```
int foo [5] = {16, 2, 77, 40, 12071};
foo[6] = 10;
cout << foo[6] << endl;
```

Accessing Array Values

- What just happened?!?

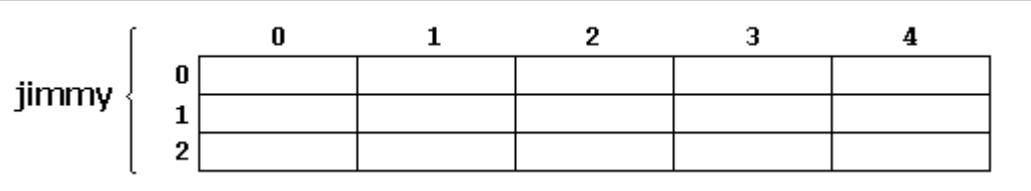




Multidimensional Arrays

- Arrays of arrays

```
int jimmy [3][5];
```



- Just a programming convenience – data is still stored contiguously in memory – the following are stored the same way:

```
int jimmy [3][5];
int jimmy [15];
```

Passing Arrays as Parameters

- You can pass an array as a parameter to a function
  - Array is not copied – only a pointer to the array is passed

```
void someFunction(int arr[])
    {
        ….
    }

int myArray [40];

someFunction(myArray);
```

Passing Arrays as Parameters

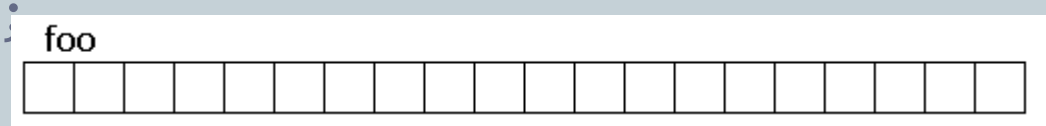
- You can pass an multidimensional arrays as a parameters also
  - First dimension is left empty

```
void someFunction(int arr[][3][4])
    {
        ….
    }

int myArray [40][3][4];

someFunction(myArray);
```

Character Sequences

- A string is really just a sequence (array) of characters

```
char foo [20];
```



- You can use this to assign values:

```
char myWord[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- Or, C++ allows you to assign a string directly during initialization:

```
char myWord[] = "Hello";
```

  - C++ will put the null character in the array automatically

- Note: this won't work in subsequent code – you'll need to assign values individually

Character Sequences

- Strings and character arrays can be used interchangeably with cin and cout
- But – arrays have a fixed size while strings have no defined size

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 << "!\n";
  return 0;
}
```

```
What is your name? Homer
Where do you live? Greece
Hello, Homer from Greece!
```

Character Sequences

- You can convert between the two:

```
char myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to string
cout << mystring;          // printed as a library string
cout << mystring.c_str();  // printed as a c-string
```

Pointers

- When you declare a variable, and run the program, the variable is placed in some location in memory
- That memory location has an address
- Memory can be addressed byte by byte, and each subsequent location is one higher than the last
  - For example, address 1776 will be followed by location 1777
  - Let's say you declare a variable:

```
int year = 2018;
```

  - You can find it's address using the address operator, &:

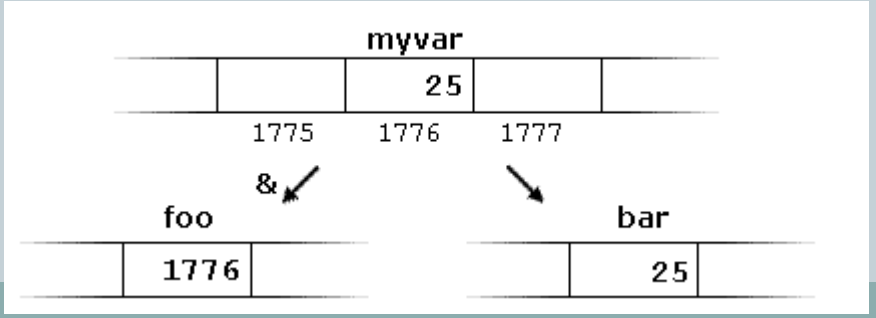
```
int * yearAddress = &year;
```

  - The * (dereference) operator says I am a pointer, and I expect to hold an address

Pointers

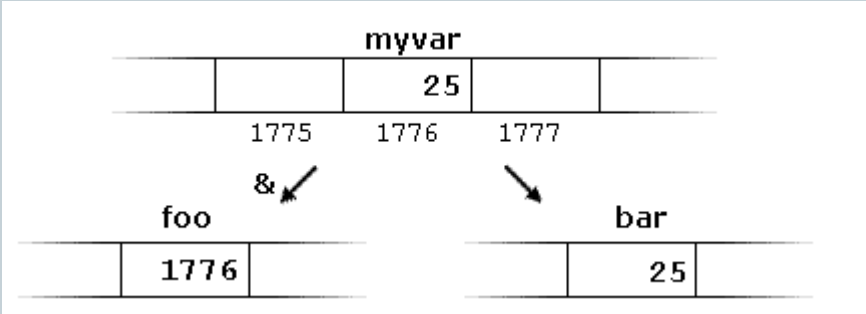
- When a variable is declared, you tell the program its data type
  - This tells the compiler how much memory is needed to store that piece of data
  - So, an int is guaranteed to be at least 16 bits, or two bytes
    - It would be stored in two consecutive memory locations
  - A pointer "points to" the variable whose address it stores

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

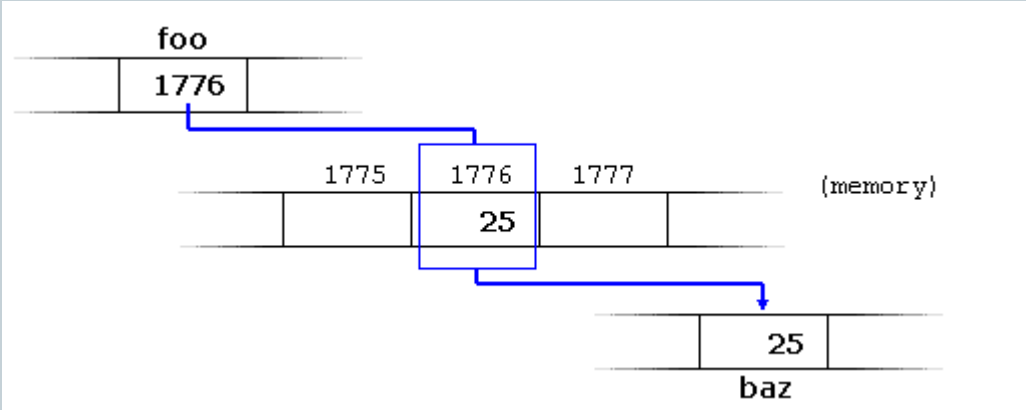


Dereference Operator

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




```
baz = *foo;
```




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

Pointers

- Reference (address) operator (&) and dereference operator (*) are complimentary
  - & can be read as “address of”
  - * can be read as “value pointed to by”
- Since a pointer can refer to the value it is pointing to, it needs to know the type of the value so it knows how much memory it occupies
  - To declare a pointer:

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

- Even though pointer point to different data of different sizes, a pointer is the same size (it always holds an address)

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 to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

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

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

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,
```

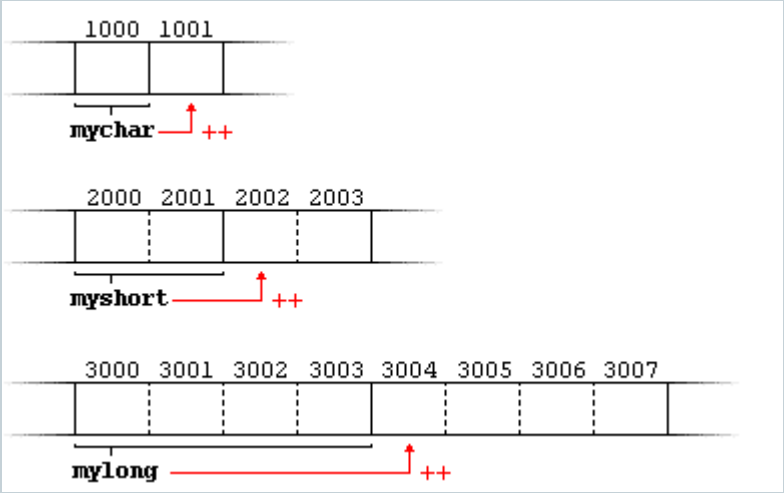
```
a[5] = 0;        // a [offset of 5] = 0
*(a+5) = 0;      // pointed to by (a+5) = 0
```

The name of an array is really just a pointer to the first address where the array is stored in memory, and the value in brackets [] is just an offset to that location

Pointer Arithmetic

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

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



Safe Pointers

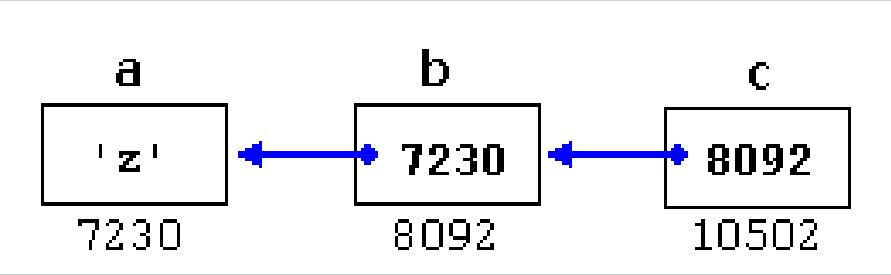
```
int x;
int y = 10;
const int * p = &y;
x = *p;           // ok: reading p
*p = x;           // error: modifying p, which is const-qualified
```

```
// 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
```

Pointers to Pointers

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



void Pointers

- void pointers are not null – they point to a value that has no type
  - Which really means they can be used to point to any data type
  - But, since the pointer doesn't know the size of the data it is pointing to, they can't be used for dereferencing
  - Need to do more work to get at the data pointed to

```
// 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
```

Invalid Pointers and Null Pointers

- Invalid pointers

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

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

- Null pointers

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

```
int * r = NULL;
```

- Null pointer is *not* the same as a void pointer!!

Pointers to Functions

- You can pass a function as a parameter to another function!

```
// 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);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

```
8
```

Summary

- Arrays
- Character Sequences
- Pointers