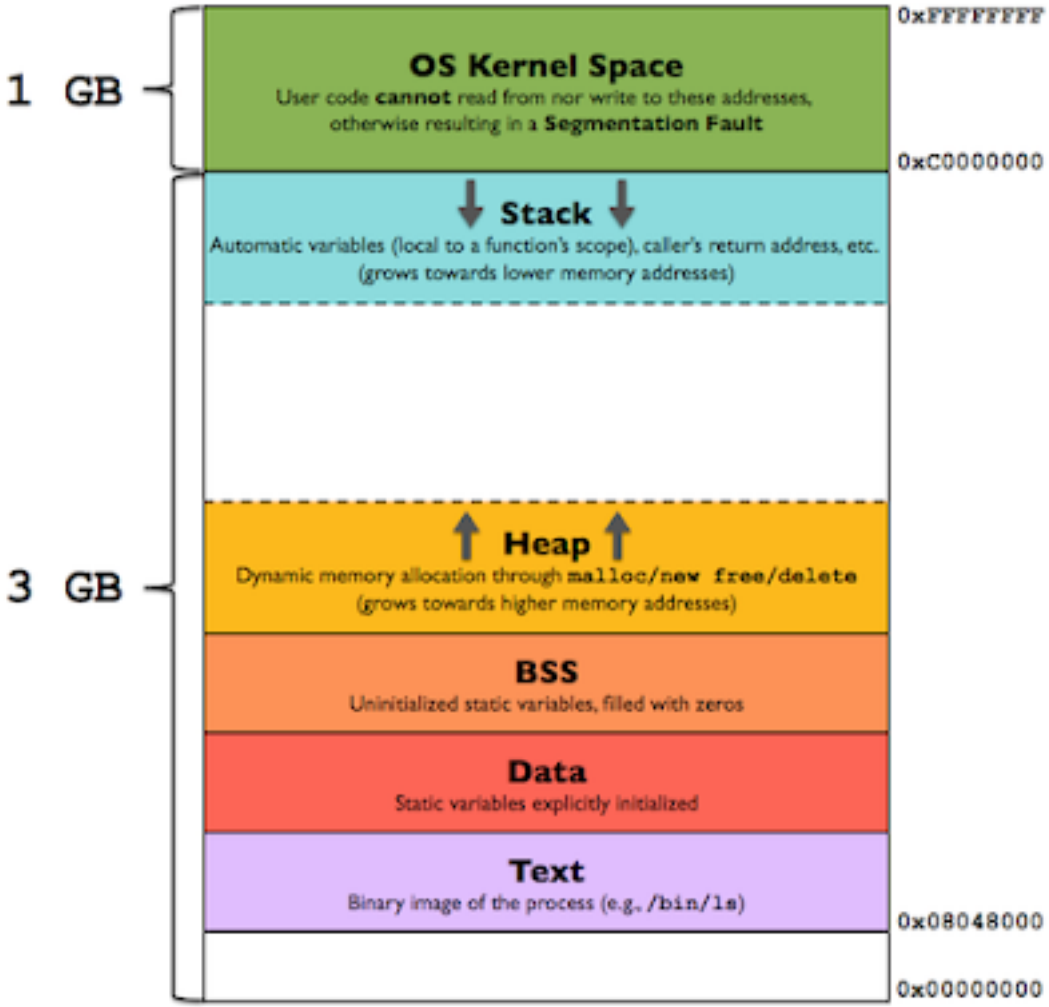
CSCI 2270 – CS 2: Data Structures

Reminders

Topics

- Memory
  - Stack vs. Heap
  - Static vs. Dynamic Memory Allocation
- Pointers

In-Memory Layout of a Program



Stack Memory

- A “stack frame” is created when a function is called. Henceforth, all the local variables of that function are created within the confines of this stack frame.
- When the function returns, its stack frame is “deleted”. The deletion of all variables happens “automagically.’
- Programmer does not need to be concerned with creating or deleting copy of variables. The run-time system provides this facility.
- A big limitation of stack: how to store variables that one can access across function calls?
  - Such a memory to store global variables is called the heap memory

Statically-Declared Arrays

```
int main(int argc, char* argv[])
{
   int ia[3];  //Array of 3 ints with garbage values
   cout << ia[1] << endl;
   float fa[] = {1, 2, 3}; //Array of 3 floats
                           //initialized: size automatically
                           //computed
   cout << fa[2] << endl; // Read different values
   return 0;
}
```

- Static Array storage is contiguous
- Array bound must be a constant expression

Stack Memory

- The stack grows and shrinks as functions push and pop local variables.
- There is no need to manage the memory yourself, variables are allocated and freed automagically.
- The stack has size limits. (Check yours with ulimit -a and set with ulimit -s 33333.
- The stack variables only exist while the function that created them, is running.

Stack Memory Limit

```
int main(int argc, char* argv[]) {
   int big[1000000]; // Trying to allocate a huge array
                     // on stack
   // Since stack memory is limit (see ulimit -a), it
   // will result in segfault
   return 0;
}
```

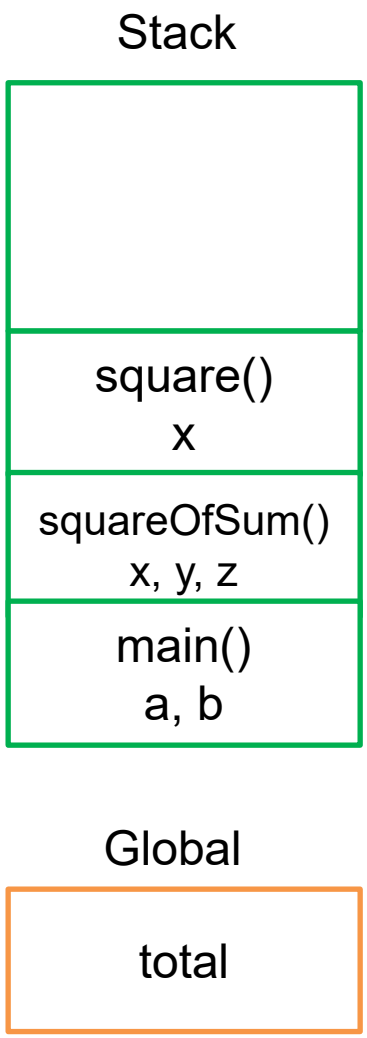
Stack Memory

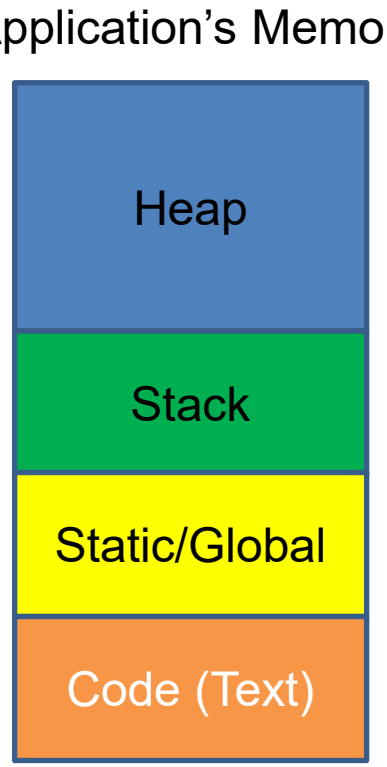
```
int total;

int square(int x)
{
   return x*x;
}

int squareOfSum(int x, int y)
{
   int z = square(x+y);
   return z;
}

int main()
{
   int a = 4, b = 8;
   total = squareOfSum(a, b);
   cout << total << endl;
}
```





Pointers

- Three data types in C++:
  - Simple: integer, bool, float
  - Structured: class, struct, enumeration
  - Pointers
- A **pointer** is a variable whose content is an address (that is, a memory address)
- The value of a pointer variable is an address
  - The data is stored in this memory space (address)
- Use * to define a pointer

```
dataType *identifier;
int *p;   // p is a pointer var that points to a
          // memory location of type int
char *ch;   // ch is a pointer var
```

Pointers

- These are equivalent:

```
int *p;
int* p;
int * p;
```

- In this example, only p is a pointer variable

```
int* p, q;
```

- The advised method is to attach the character * to the variable name:

```
int *p, q;
```

- Here, we have 2 pointers:

```
int *p, *q;
```

Pointers

- **Remember! The value of a pointer is a memory address!**

Pointers

- `&` is called the **address of operator** and returns the address of its operand

```
int x;
int *p;
p = &x;      // assigns address of x to p
             // x and the value of p refer to the same
             // memory location
```

Pointers

- * is referred to as the **dereferencing operator** or **indirection operator**
- * refers to the object to which the operand of the * (that is, the pointer) points

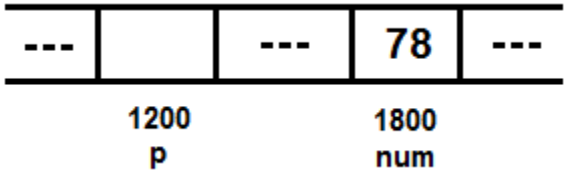
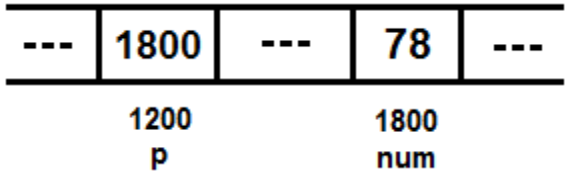
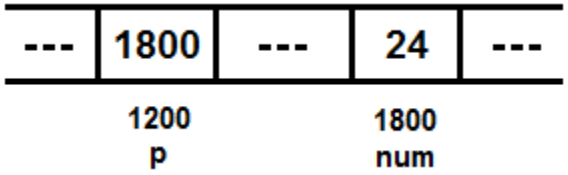
```
int x = 25;
int *p;
p = &x;                    // store the address of x in p

cout << *p << endl;  // prints value stored in the memory space to
                           // which p points, which is the value of x  (i.e. 25)
```

- `*p = 55;` // stores 55 in the memory location to which p points –
  // that is, 55 is stored in x

Pointers

```
1. int *p;
2. int num;
3. num = 78;
4. p = &num;
5. *p = 24;
```

After statement	Values of the variable	Explanation
3		The statement num = 78; stores 78 into num.
4		The statement p = # stores the address of num, which is 1800, into p.
5		The statement *p = 24; stores 24 into the memory location to which p points. Because the value of p is 1800, statement 5 stores 24 into memory location 1800. Note that the value of num is also changed.

Pointers

- If you don't want a pointer to point to anything, set it to NULL:

```
p = NULL;
```

- The statement above is equivalent to:

```
p = 0;
```

Note: 0 is the only number that can be directly assigned to a pointer variable.

- Note: setting pointers to NULL is useful in avoiding dangling pointers (more on this later).
- Note: It is an error to dereference an uninitialized pointer or the NULL pointer. We will get a memory fault!

Pointers - Operations

- Operations on Pointer Variables

```
int *p, *q;
p = q;          // copies the value of q and p

// Both p and q point to the same memory location.
// Any changes made to *p automatically change the
// value of *q, and vice versa.
```

Pointers and Classes

```
string *str;                    // pointer to a string
str = new string;               // allocates memory of type string and stores the
                                // address of the allocated memory in str
*str = "Sunny Day";             // stores the string in the memory to which str points
```

Assume we want to find the length of the string via the length() function.

`(*str).length()` returns the length of the string.

Sometimes, programmers mess up the syntax above that uses the dereferencing operator, so instead we can use the equivalent expression below, which contains the **member access operator arrow**, ->:

```
str->length()
```

Use the syntax above for accessing class (struct) components!

Pointers to Struct Objects

- A structure pointer is a type of pointer that stores the address of a structure typed variable.

```
struct Animal
{
   int age;
   float weight;
};

int main()
{
   // creating a Complex structure variable
   Animal var1;
   Animal *ptr = &var1;

   var1.age = 5;
   var1.weight = 0.33;
   // accessing values of var1 using pointer
   cout << “Age:  << ptr->age << endl;
   cout << “Weight: “ << ptr->weight << endl;
}
```

Pointers to Struct Objects

```
struct Distance
{
    int feet;
    float inch;
};
```

```
int main()
{
    Distance *ptr;
    Distance d;
    ptr = &d;

    cout << "Enter feet: ";
    cin >> (*ptr).feet;
    cout << "Enter inch: ";
    cin >> (*ptr).inch;

    cout << "Displaying information." << endl;
    cout << "Distance = " << (*ptr).feet << " feet "
                          << (*ptr).inch << " inches";
    return 0;
}
```

Pointers

- Up to this point, we have used pointers to manipulate data only into memory spaces that were created using other variables. In other words, the pointers manipulated data into existing memory spaces.
- So, let's learn about **dynamic variables**. We're going to allocate and deallocate memory during program execution using pointers.
- Use `new` and `delete` operators to create and destroy dynamic variables, which are reserved words in C++.

Heap Memory (Dynamic)

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
  - To allocate memory on the heap, you must use operator new.
  - Once you have allocated memory on the heap, you are responsible for deleting it using delete operator to deallocate that memory once you don't need it any more.
- How the heap is managed is really up to the runtime environment!
- If you don't deallocate memory, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).
- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.
- Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap.

Heap Memory

- Let's look at what we've learned thus far

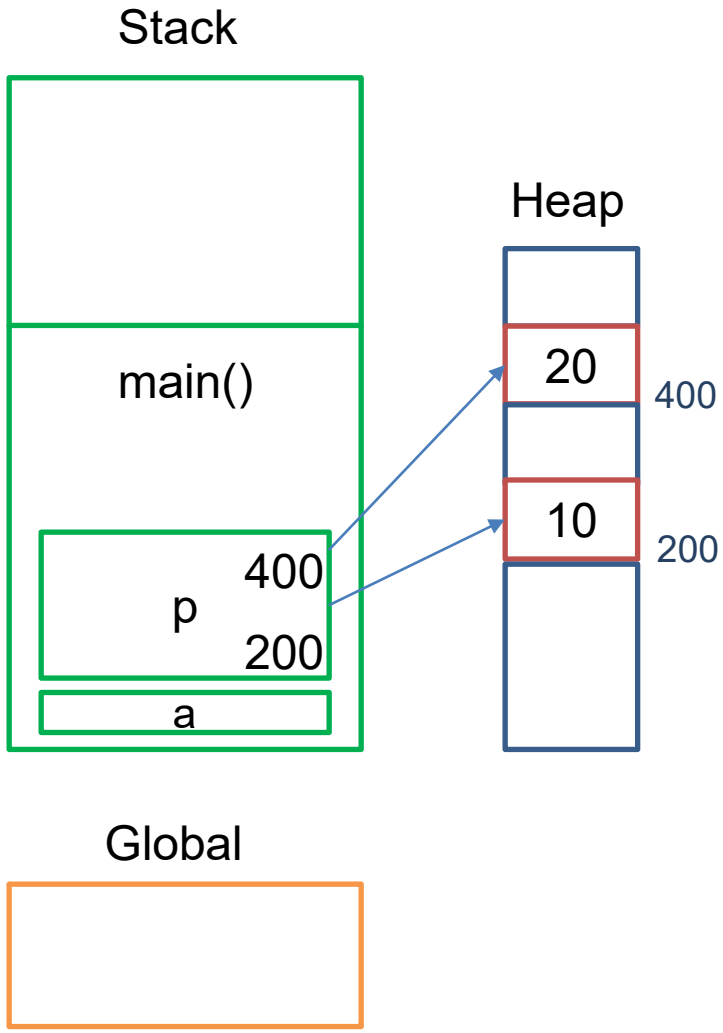
```
int *p;
int x;
p = &x;     // stores the address of x in p. However, no new memory is allocated!
```

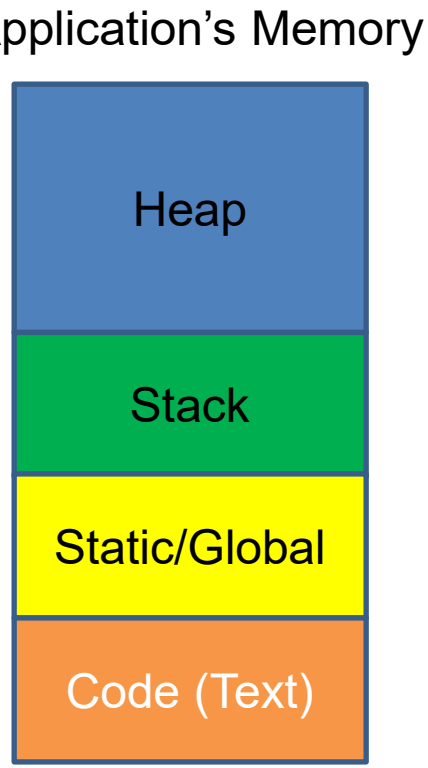
- Let's create a variable during program execution somewhere in memory (heap)
- In order to use the heap in C++, we use `new` and `delete` keywords
- Use a pointer to allocate memory on the heap

```
int *p;          // p is a pointer of type int
p = new int;     // enough memory to store one int type variable and stores the
…                // address of the allocated memory in p
*p = 28;         // stores 28 in the allocated memory
delete p;        // free the memory back to the heap
                 // note: "delete" does not delete the pointer (can re-use the pointer)
```

Heap Memory

```
int main()
{
   int a;   // goes on stack
   int *p;
   p = new int;
   *p = 10;
   delete p;
   p = new int;
   *p = 20;
   delete p;
}
```





Heap Memory

- Note: The operator `new` allocates memory space of a specific type and returns the (starting) address of the allocated memory space. However, if the operator `new` is unable to allocate the required memory space (for example, there is not enough memory space), the program might terminate with an error message.

Heap Memory Limit?

```
int main(int argc, char* argv[]) {
    int *big = new int[10000000]; // Trying to allocate
                                  // a huge array on heap
    // No problem whatsoever!
    return 0;
}
```

Dynamically-Declared Arrays

```
int main(int argc, char* argv[])
{
    int *pa = 0; // pa is a pointer to integers
    int n;
    cout << "Enter dynamically allocated array size:";
    cin >> n;
    pa = new int[n]; // e.g. if n=10, 10 contiguous memory locations
    for (int i = 0; i < n; i++) {
       pa[i] = i;
     }
     // Use pa as a normal array
    delete[] pa; // When done, free memory pointed to by pa.
    pa = 0; // Clear pa to prevent using invalid memory reference
            // (or use NULL)
    return 0;
}
```

Stack vs Heap

- Stack:
  - very fast access
  - don't have to explicitly free variables
  - space is managed efficiently by CPU,
  - memory will not become fragmented
  - local variables only
  - limit on stack size (OS-dependent)
  - variables cannot be resized
- Heap
  - variables can be accessed globally
  - no limit on memory size
  - (relatively) slower access
  - no guaranteed efficient use of space, memory may become
  - fragmented over time as blocks of memory are allocated, then freed
  - you must manage memory (you're in charge of allocating and freeing variables)

Dangling Pointers

Dangling Pointers - Example

Memory Leak

- Note: If you forget to destroy a dynamic variable that is no longer needed, you will end up with a memory leak. Use `delete` to destroy dynamic variables.

- If you don't delete it, the memory space remains marked as allocated. In other words, it cannot be reallocated. It's an unused memory space that cannot be allocated.

- The dangerous part comes when we forget to delete dynamic variables that execute, let's say, thousands of times. The program might run out of memory space for data manipulation, and eventually result in an abnormal termination of the program.

Pointers vs References

- References are less powerful than pointers. References in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have above restrictions, and can be used to implement all data structures. References are more powerful in Java, which is the main reason Java doesn't need pointers.
- References are safer and easier to use.
  - Uninitialized pointers = possible seg fault
  - Memory leaks
- Pointers can be assigned to NULL. References cannot.
- Pointers can point to pointers. References only have one level of direction. A reference is always initialized once and only once, which means it cannot be changed to refer to another variable or memory location. Pointers can be changed at will.
- References are useful for function argument lists and function return values.

Questions?