Dynamic Memory

Review

- C++ must figure out the amount of space each variable takes up in memory at compile-time (before the program is run).
- When a function is called, C++ reserves a block of memory for ***all*** of that function's variables at once.
- Therefore, C++ always knows, before a program starts running, the memory address of every variable in a program, **relative to the block of memory for the function that variable belongs to.**

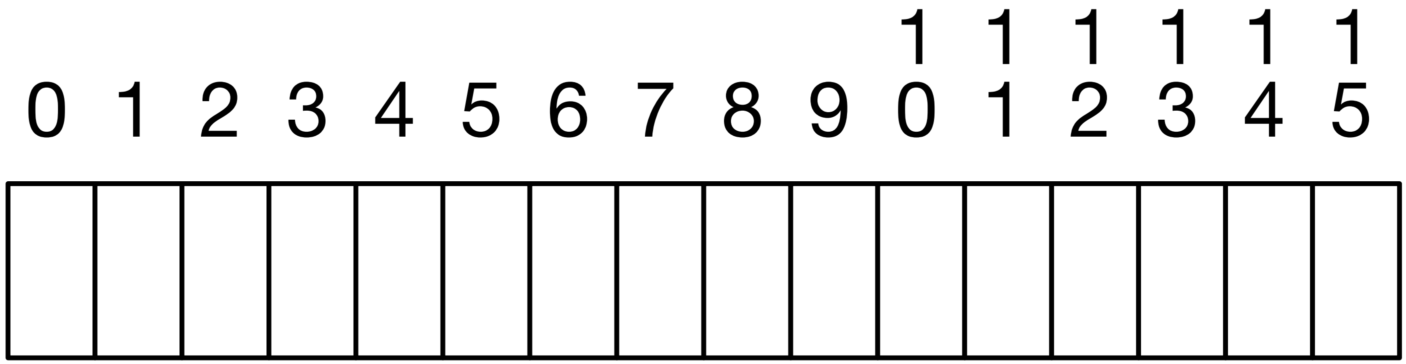
```
int main()     // main needs 8 bytes
{
  int x;       // 4 bytes
  int y;       // 4 bytes
  f();
  g();
}
void f() {     // f needs 4 bytes
  int z;
}
void g() {     // g needs 4 bytes
  int q;
  f();
}
```

```
int main()    // main needs 8 bytes
{
  int x;      // 4 bytes  (start of block + 0)
  int y;      // 4 bytes  (start of block + 4)
  f();
  g();
}
void f() {    // f needs 4 bytes
  int z;      // 4 bytes  (start of block + 0)
}
void g() {    // g needs 4 bytes
  int q;      // 4 bytes  (start of block + 0)
  f();
}
```

- Why does C++ care about memory addresses relative to a function's block of memory?
- If C++ knows:
  - the starting address for a function's block of memory, and
  - the relative offset for every variable in that function
- then C++ can very quickly compute the memory address for any variable by adding those two pieces together.

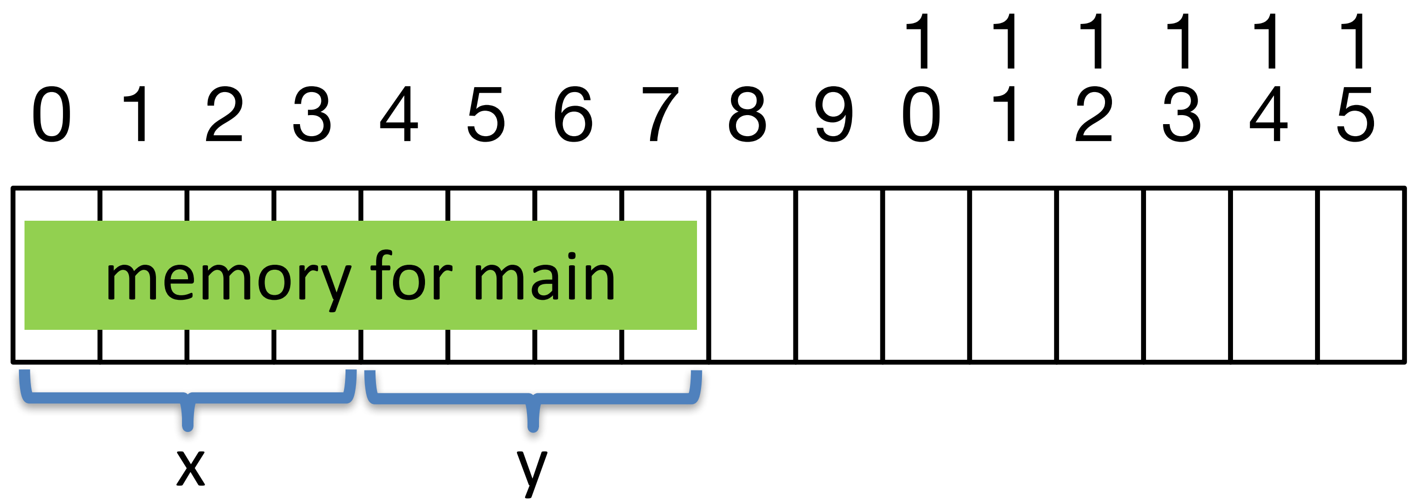
Before program begins

```
int main()
{
  int x;
  int y;
  f();
  g();
}
void f() {
  int z;
}
void g() {
  int q;
  f();
}
```



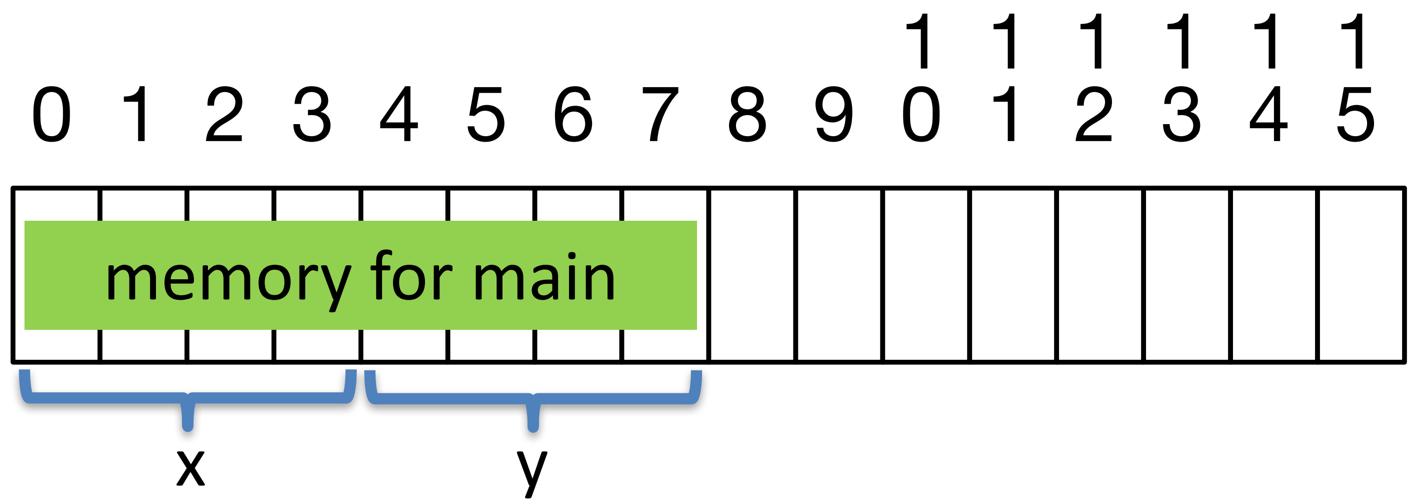
main() is called

```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```



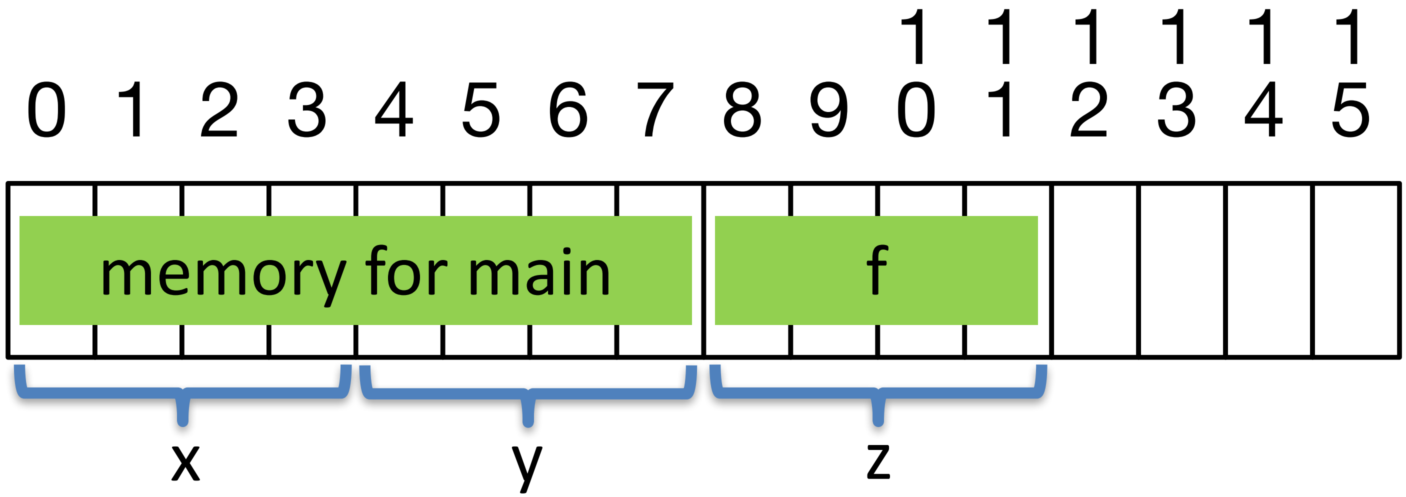
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

f() is about to be called



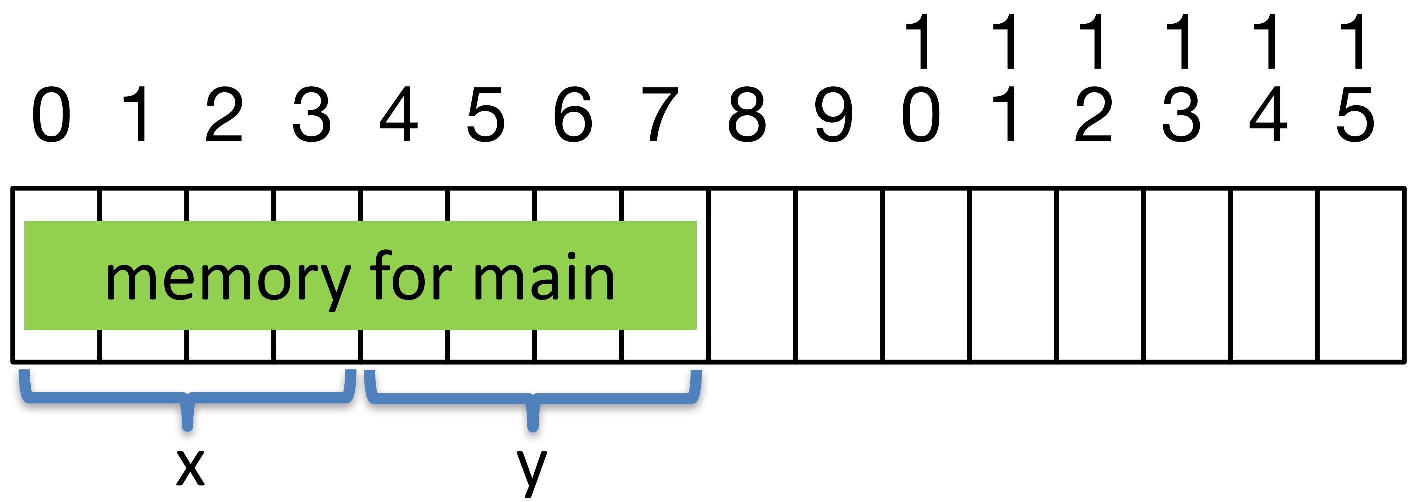
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

f() is called



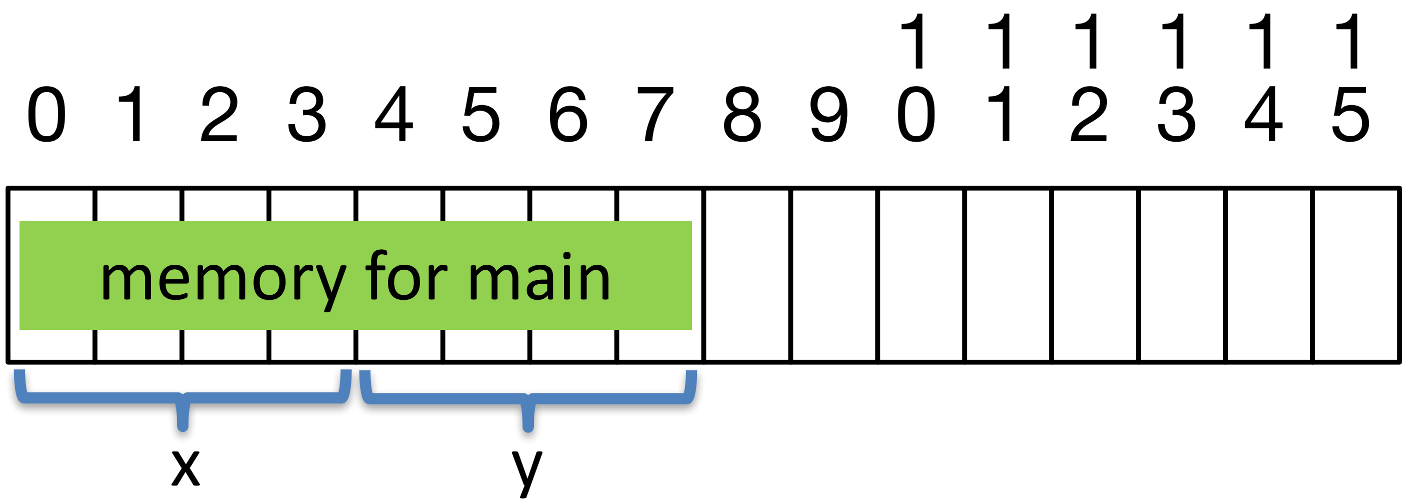
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

f() finishes; go back to main()



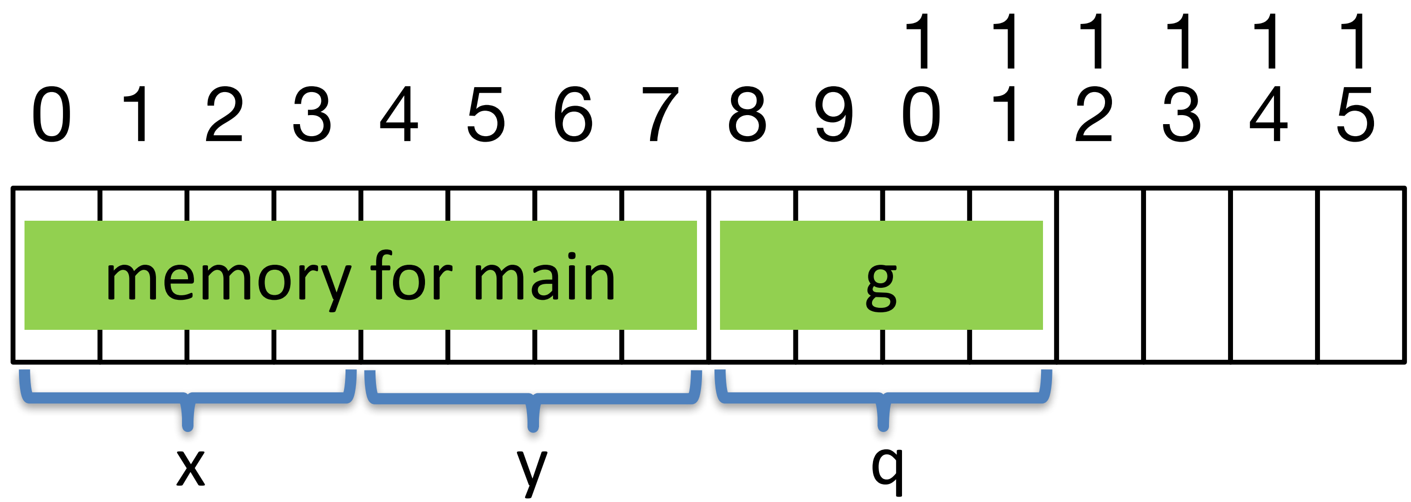
g() is about to be called

```
int main()
{
  int x;
  int y;
  f();
  g();
}
void f() {
  int z;
}
void g() {
  int q;
  f();
}
```



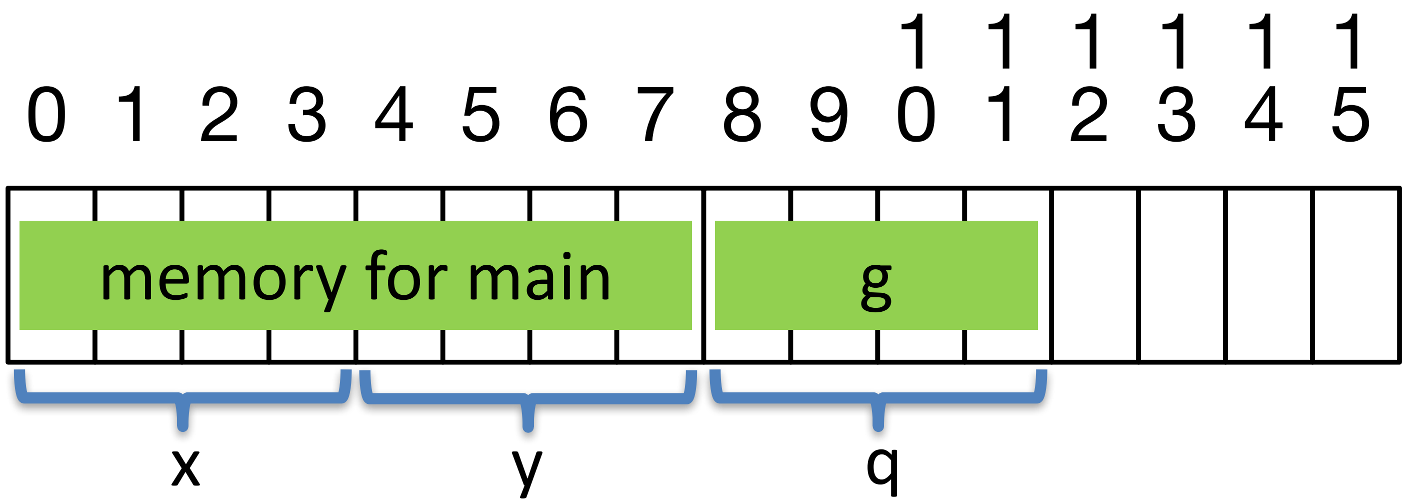
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

g() is called



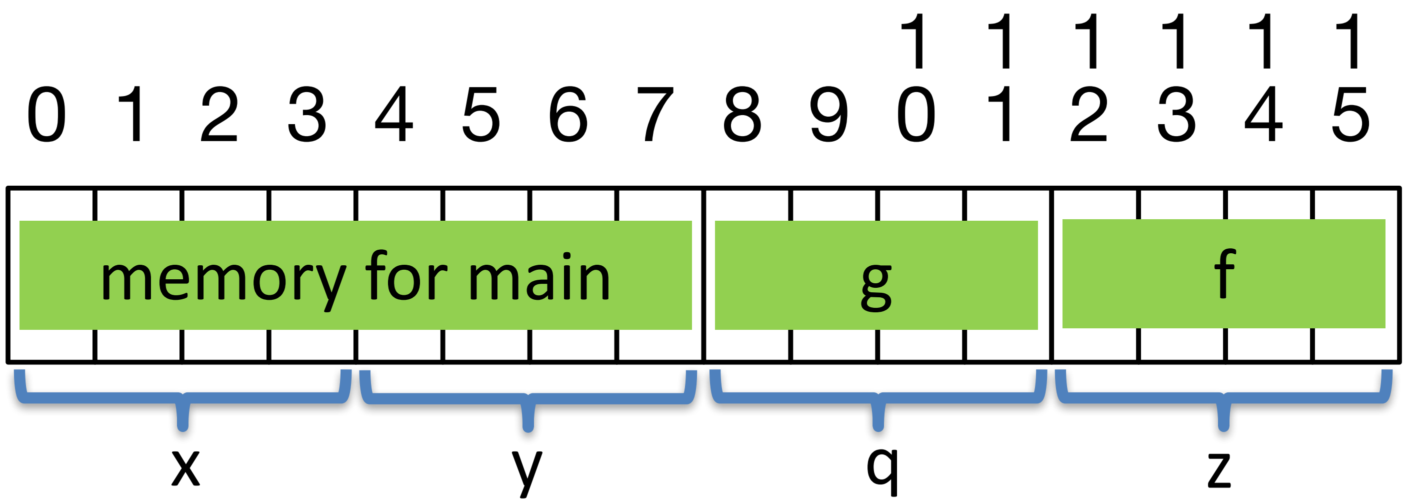
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

f() is about to be called from g()



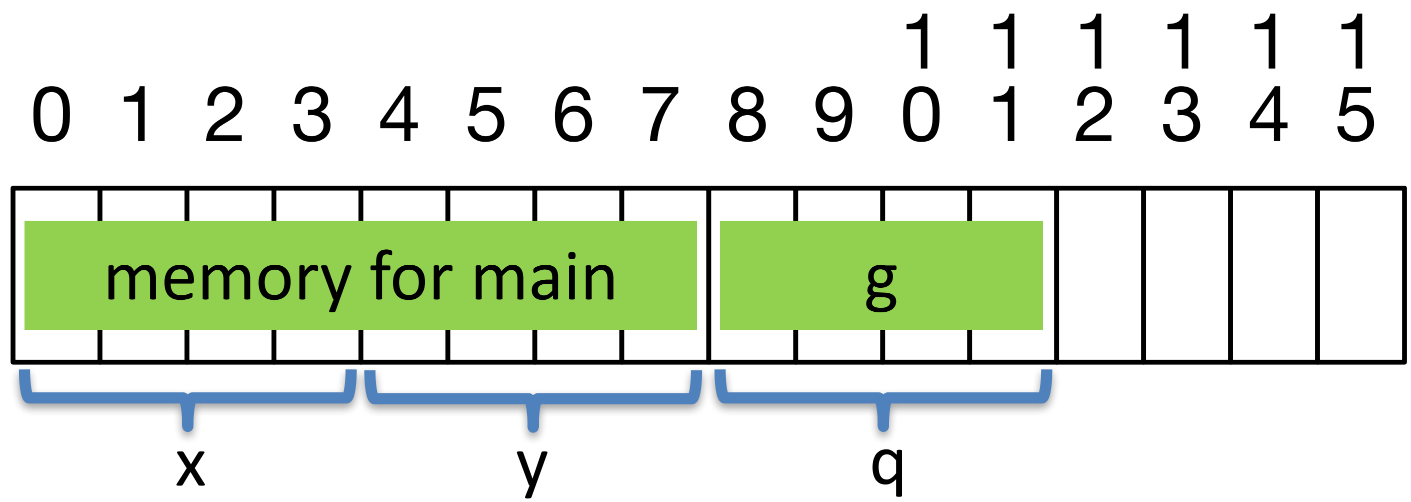
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

f() is called from g()



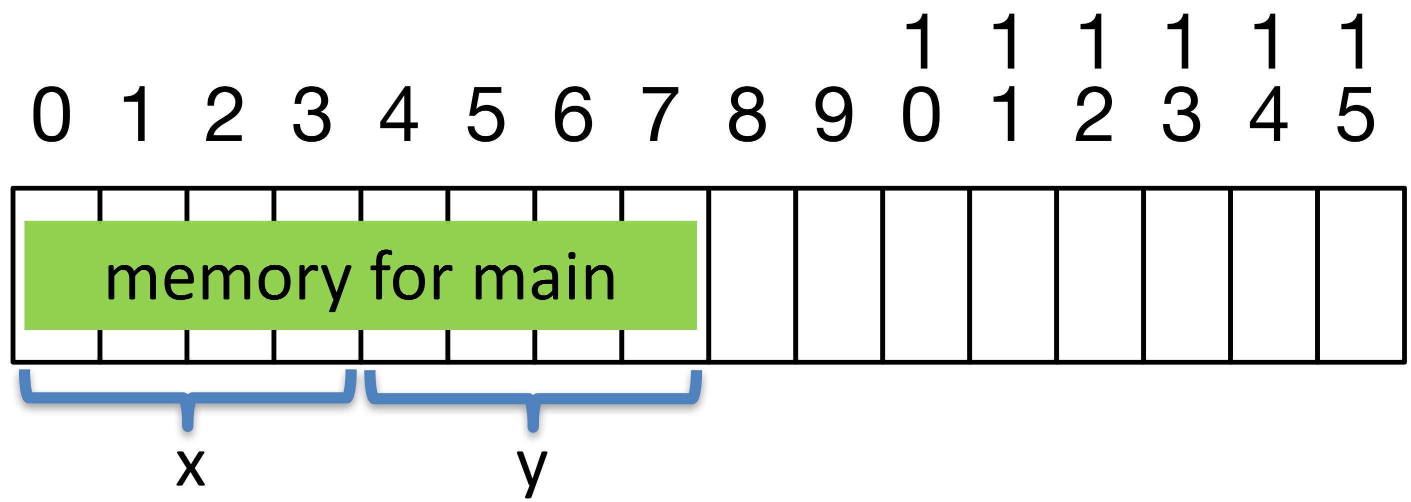
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

f() finishes running; go back to g()



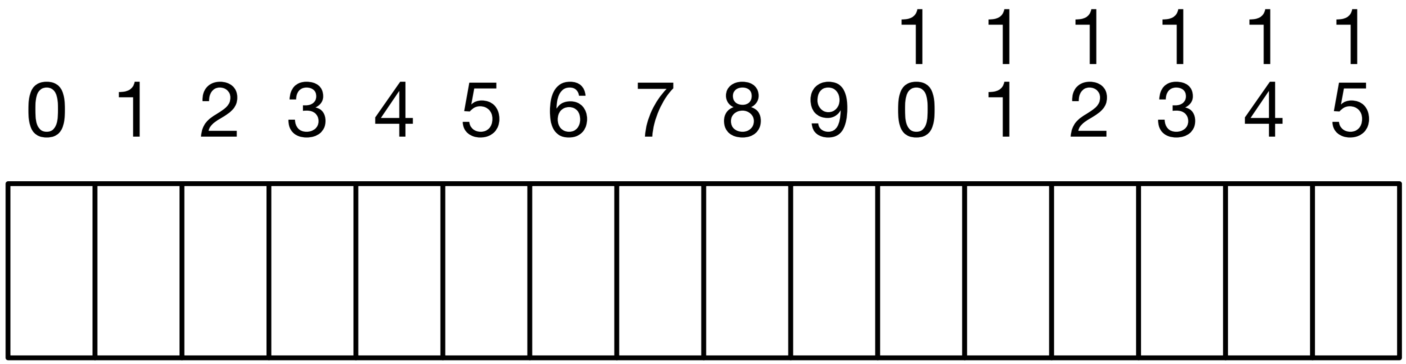
```
int main()
{
   int x;
   int y;
   f();
   g();
}
void f() {
   int z;
}
void g() {
   int q;
   f();
}
```

g() finishes running; go back to main()



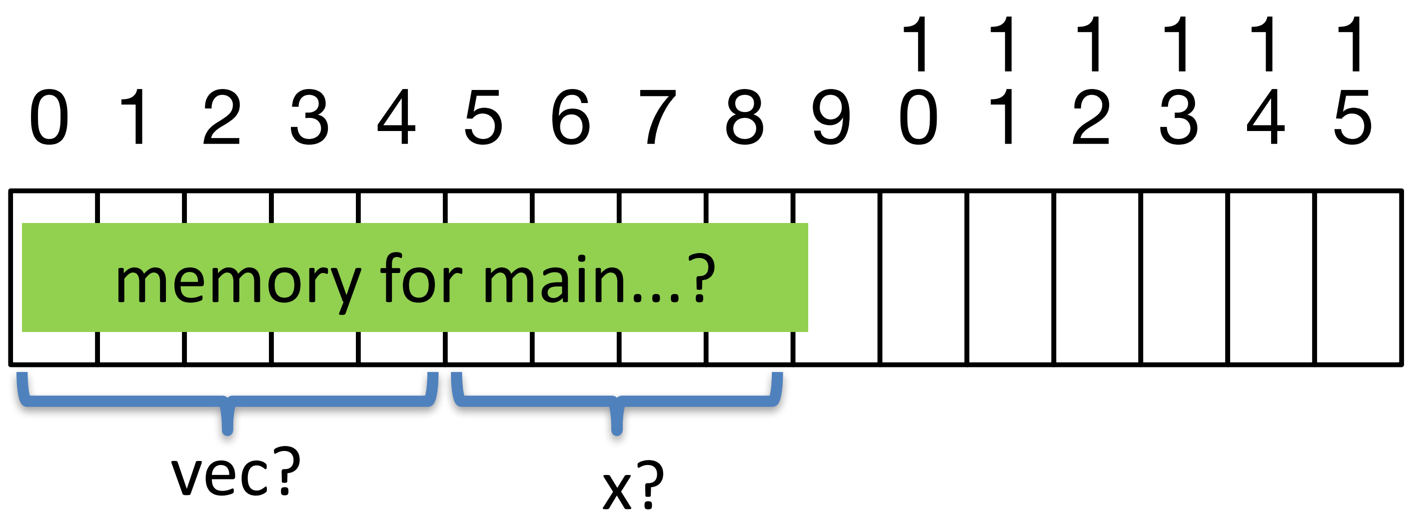
```
int main()
{
  int x;
  int y;
  f();
  g();
}
void f() {
  int z;
}
void g() {
  int q;
  f();
}
```

main() finishes



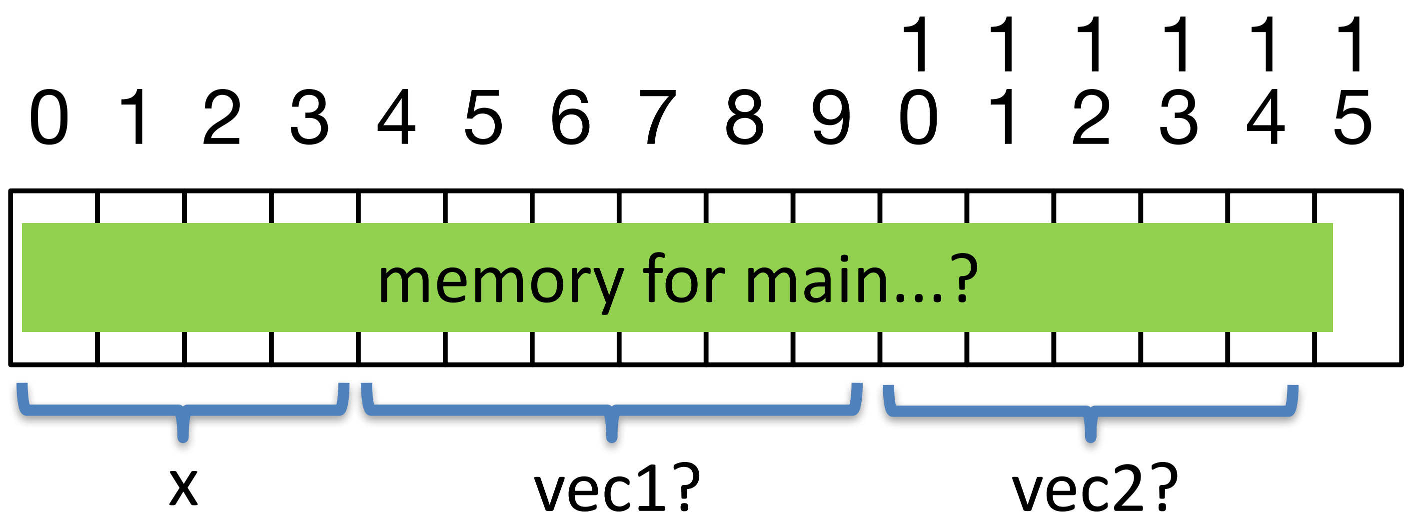
But what about vectors?

```
int main() {
  vector<int> vec;
  int x;
}
```



But what about vectors?

```
int main() {
   vector<int> vec1, vec2;
   int x;
}
```



C++ has two areas of memory

- The "regular" area of memory that C++ uses is called the ***stack***.
  - This is where C++ puts variables that it knows the size of at compile-time because they have ***fixed sizes*** (ints, doubles, etc).
  - Variables on the stack are automatically allocated memory when their functions are called, and automatically deallocated when their functions end.
  - Therefore, sometimes they are called ***automatic variables.***

- There is a second area of memory that ***must*** be used for storing variables whose sizes cannot be determined at compile time (strings, vectors, etc).
  - This area is called the ***heap.***
- Variables on the heap are not automatically allocated memory, nor is their memory ever automatically deallocated (opposite of stack variables).
- The programmer explicitly controls when the memory is allocated and deallocated.

Why is this useful?

- Create variables that may grow and shrink in size as necessary.
- Create more sophisticated data structures.

Dynamic memory allocation

- All access to heap variables is done through pointers.
- `type *ptr = new type;`
  - allocate memory on the heap for one new variable with the given *type* and return a pointer to it.
- `delete ptr;`
  - deallocate the memory pointed to by ptr
  - good idea to then set `ptr` to `nullptr`
- You must deallocate all your memory when you are done with it!

Dynamic memory gotchas

- For automatic (stack) variables, you normally have two ways to access the variable: the variable itself and any pointer(s) to the variable.
- For heap variables, the only access is through a pointer.

Dynamic memory gotchas

- The ***pointer*** to the dynamic memory is still an automatic variable, so it can be passed and returned from functions like normal.
  - Treat the ***pointer variable*** like any other variable.
  - Treat the ***memory it points to*** differently!
- You can copy that pointer as much as you want, but you must `delete` it exactly once (no matter how many copies there are floating around).

Dynamic memory gotchas

- After heap memory is `deleted`, it may be allocated for something else, so any existing pointers to that memory should be considered invalid.
- Deleting the same memory twice is bad.
- You can delete memory anytime you want.

- Allocate two new ints on the heap (dynamically). (keyword is **`new`**)
- Set them equal to 10 and 20 and print them.
- Switch the pointers so each pointer now points to the opposite int.
- Print them again.
- Deallocate the memory. (keyword is **`delete`**)

- Optional: experiment with deleting something that has already been deleted. What happens? What happens if you assign to something that has already been deleted?

Allocating lots of variables at once

- `type *ptr = new type[num];`
  - allocate memory on the heap for **num** new variables of `type` and return a pointer to them.
  - Use square bracket [] syntax to access each element (like a vector, but no size/push_back).
- **`delete[]`** `ptr;`
  - deallocate the memory pointed to by ptr
  - only use delete[] with new[]
  - only use delete with new

Variables that grow and/or shrink

- Using **`new`** `type[num]` still doesn't make the dynamic memory grow or shrink.
- So how do vectors work?
  - A vector starts off my allocating (using new) a "default" amount of space for items in the vector.
  - If we add too many things to a vector, it will allocate more space, copy everything in the vector into the new space, then delete[] the old space.

- Allocate (on the heap) an array of 3 doubles.
- Assign some numbers to the array.
- [Pretend that we want to add more numbers.]
- Allocate (on the heap) a second array of 6 doubles.
- Copy the doubles from the old array into the new one.
- delete[] the old array.
- Print the new array.
- delete[] the new array.