Programs, files, and memory

- once we have compiled a program, its executable code is sitting somewhere in the file system
- when we run the program that executable code is copied into the computer's memory, and space is established in memory for the global variable and constants, as well as space for the main routine's local variables and constants
- as functions are called, space in memory is also set aside for their parameters, local variables, and local constants
- as the function calls complete, their space in memory is released so it can be used later for other function calls

Pointers and memory addresses

- every variable in our program is stored somewhere in memory
- different types of data require different amounts of storage, e.g. 1 byte for a char or bool, 4 bytes for an int or float, 8 bytes for a long or double, etc
- memory is divided into tiny storage locations, each a single byte in size, each storage location is given a unique integer address
- for a gigabyte of memory these addresses might go from 1, 2, 3, etc to 1073741823
- when we talk of the “address of a variable” we mean the address of the first of its N consecutive bytes in memory

Addresses and hex notation

- as memory tends to be divided into blocks that are powers of 2, memory patterns are often easier to spot when shown in a base that is a power of 2, instead of base 10
- the two most commonly used are base 16 (hexadecimal) and base 8 (octal)
- we'll use base 16 for showing memory addresses
- to do so, values 0..9 are represented normally, but then 10 is represented as A, 11 as B, ... 15 as F
- thus the hex digits are 0123456789ABCDEF

Quick look at hex

- if we have 537 in decimal we know this represents $5*10^2 + 3*10^1 + 7*10^0$
- in hex the base is 16, so 10 in hex represents the (decimal) value $1*16^1 + 0*16^0 = 16$
- similarly123 represents $1*16^2 + 2*16^1 + 3*16^0 = 256 + 32 + 3 = 291$
- 2B3F would represent $2*16^3 + 11*16^2 + 3*16^1 + 15*16^0 = 8192 + 2816 + 48 + 15 = 11071$
- in C++, integers in decimal are written normally, and when a number is in hex it is preceeded by 0x, thus 0x0123 == 291

Looking up sizes and addresses

- We can look up the amount of storage needed for a given datatype with the sizeof function

```
cout << sizeof(char); // displays 1
cout << sizeof(int); // displays 4
cout << sizeof(double); // displays 8
```

- we can look up the memory address of a variable (in hex) using &

```
int i;
cout << &i;
// displays something like 0x7fff9361a080
```

Pointers & and *

- pointers: variables whose purpose is to store or use memory addresses (they point to some spot in memory)
- pointers also specify what kind of data they can point at, e.g. pointer to an int, pointer to a float, pointer to a string
- we declare pointer variables by adding * to the data type

```
int x = 10; // regular integer variable
int *iptr;  // iptr a pointer for ints
iptr = &x; // iptr holds x's memory address, points to x
```

Dereferencing pointers with *

- we can use a pointer to access (read or alter) the contents of memory at that location
- this is done through the use of *, and is called dereferencing

```
int x = 10;
int* iptr = &x;
// next, use * to get at memory through the pointer
*iptr = 20; // put 20 in memory where iptr says, overwrites x
cout << x;  // displays 20
```

Passing pointers

- a function can change the value of a variable if we pass the function a pointer to the variable, e.g.

```
void overwrite(int* ptr) { // expects a pointer to an int
    cout << “enter an integer”;
    cin >> (*ptr); // change memory where the pointer says
}
int main() {
      int y;
      overwrite(&y); // pass the address of y
}
```

- this is what the compiler really does to your code when you use pass by reference!

You've already used this...

- now scanf's use of & finally makes sense
- scanf was written in C, which didn't have pass by reference and thus had to use pointers

```
int x;
scanf(“%d”, &x); // pass address of x, so scanf can change x
```

Arrays and pointers

- An array variable acts like a (constant) pointer to the first element of the array

```
int array[10];
// all three of the following will give the same address!
cout << array;
cout << &array;
cout << &(array[0]);
```

- thus when we pass an array as a parameter to a function we're really passing a memory address, which is why functions can change the contents of arrays

Structs and pointers

- we can use & to look up the addresses of struct variables and the individual fields within them, e.g.

```
struct point {
    int x, y;
};
int main() {
    point p = { 10, 20 };
    cout << &p << “,” << &(p.x) << “,” &(p.y) << endl;
    // address of p and p.x will be the same
}
```

Memory address of other things

- we can use & to look up the memory address of many things: constants, variables, parameters
- we can even use math on pointers to explore forward/backward from a memory address:

```
int* iptr = &x;
iptr += 8;
cout << (*iptr); // see what's 8 bytes away from x
```

- each program is run in its own virtual memory space, so you're not seeing the memory from other programs, but you can explore the structure of your program's memory

Memory organization

- your program's logical memory is generally structured (vaguely) as follows:
  - space for admin info for your program
  - space for your program's machine code
  - space for global constants/variables
  - a runtime stack (growing down into the free space)
  - ***the currently free/available space***
  - dynamic, or heap, storage (growing up into the free space)

The runtime stack

- each call to each function gets memory space for its parameters, local variables
- this is stored in the runtime stack (so called because each function call's space gets “stacked” on top of the space for the function that called it)
- when the function call completes its space gets cleaned off the stack

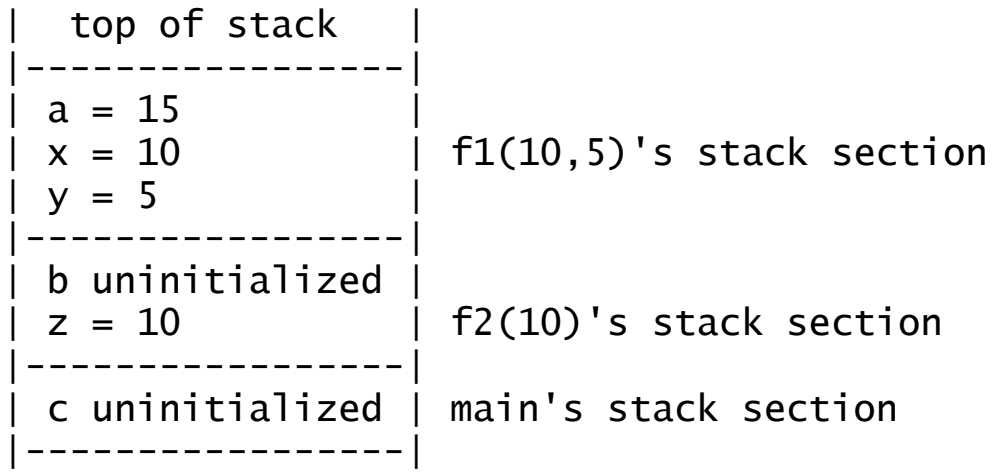
Runtime stack example

```
int f1(int x, int y) {
   int a = x + y;
   return a;
}

int f2(int z) {
   int b = f1(z, 5);
   return b;
}

int main() {
   int c = f2(10);
   cout << c;
}
```

when we're just before return a, the stack may be



when f1 returns its space will be cleaned off the stack, then when f2 returns its space will also be cleaned off

The heap: dynamic memory

- the other crucial use for pointers is dynamic allocation and deallocation of memory
- so far, all our arrays and variables have sizes that are known at compile time - the compiler knows how much space to set aside in memory for that variable
- sometimes we don't know how much space we'll need for an item until the program is already running: dynamic allocation allows a program to request memory “on the fly”
- this memory is allocated from the heap space
- dynamic allocation/deallocation is our next big topic