Part V

Memory and pointers

Outline

Pointers

- A variable `x` stored in memory is stored at a specific place in memory (its address)
- If we store that address in a variable `p`, then `p` is a pointer to `x`

```
int x = 5;
int* p = &x; // Store address of x in p
std::cout << *p << std::endl; // Prints value of x
*p = 6; // Changes x
std::cout << x; // Prints 6
```

- Note the address-of operator `&`.
- This is different from the pass-by-reference modifier `&` seen earlier.

Declaring multiple pointers

- When declaring multiple variables:

  ```
  int* a,b,c;
  ```

  creates an integer pointer `a` and *integers* `b` and `c`
- A clearer way to write the same thing is

  ```
  int *a, b, c;
  ```

  or

  ```
  int *a;
  int b,c;
  ```

- It may help to think of the second example as declaring: “A variable which, when dereferenced, gives an integer.”
- Alternatively, just avoid this confusion altogether by using the third way of writing this.

Aliasing

- Through the use of pointers and references, a single variable in memory can be changed using multiple labels
- This can cause programmer errors, if variables are changed in non-obvious ways: just because there is no `x=` doesn't mean that `x` is constant:

```
int f(int& x, int &y){
  const int a = x;
  y = y+1;
  const int b = x;
}
f(s, s); // Would lead to a != b within f
```

- Note that not even using `const` on `x` enforces this

Const-pointers

- Const-ness extends naturally to pointers
- However, some clarification may be needed:

```
const int x = 0;
int y;
const int z = 1;
const int* p; // Pointer to a constant integer
*p = 5; // Compiler—error
p = &x; // OK
p = &y; // OK — but y cannot be changed through p
int* const q = &y; // Constant—pointer to a non—constant
    integer
*q = 4; // OK — changes y
q = &z; // Compiler error — would change q

const int* const r = &x;
*r = 3; // Compile—error
r = &z; // Compile—error

// Multiple constant ptrs to const vars
const int *const s = &x, *const t = &y;
```

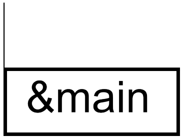
Outline

Heap and Stack

- The stack is a Last-In-First-Out structure (imagine a stack of plates)
- The stack contains the local data used by all functions currently executing
- When calling a function, its parameters are copied onto the stack, As local variables are allocated, they are also placed on the stack
- When exiting a function:
  - its return value is copied out
  - all local variables are deleted
  - and the function is taken off the stack
- This leaves the calling function at the top, so it can continue with evaluation
- The stack has limited size, so very large arrays can't be stored on it

Stack simulation

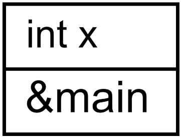
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
6

Stack simulation

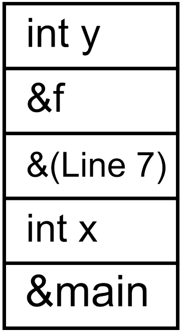
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
7

Stack simulation

```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
1

Stack simulation

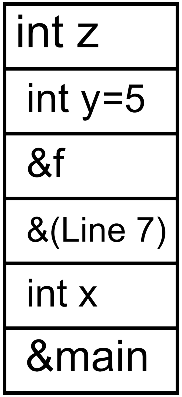
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```

int y=5
&f
&(Line 7)
int x
&main

Line Counter:
1

Stack simulation

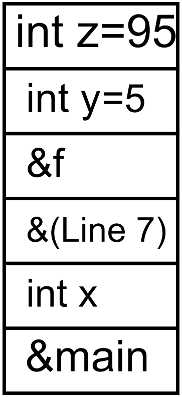
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
2

Stack simulation

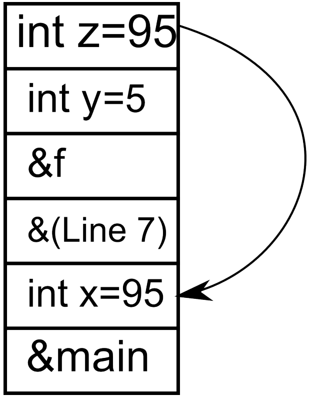
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
2

Stack simulation

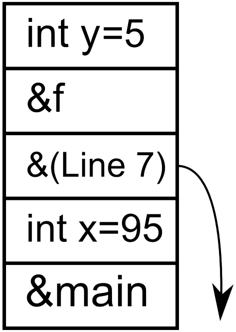
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
3

Stack simulation

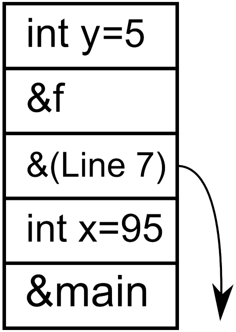
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
4

Stack simulation

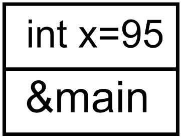
```
1  int f(int y){
2   int z = 3*y*y + 4*y;
3    return z;
4  }
5
6  int main(void){
7    int x = f(5);
8  }
```



Line Counter:
7

Stack simulation

```
1 int f(int y){
2  int z = 3*y*y + 4*y;
3   return z;
4 }
5
6 int main(void){
7   int x = f(5);
8 }
```



Line Counter:
7

Heap

- The heap is (essentially) the rest of your computer's memory
- Can store memory-blocks of arbitrary size
- Allocation/deletion of blocks can be done in any order
- Allocation has some time/memory overhead, so usually good to allocate a few large blocks of memory, rather than many small blocks.
- Blocks are not automatically deleted once their pointer goes out of scope.
- This is useful because blocks persist without being explicitly copied, saving the time taken to copy data.
- This is a problem because you have to remember to release the memory explicitly, otherwise your program's memory usage may grow too large.

Memory allocation (heap)

- We can request an amount of memory from the OS and have it return a pointer to a contiguous (uninitialized) block of memory.

```
int* p = new int; // Allocate memory for a single integer
*p = 8; //  Dereferencing
std::cout << *p << std::endl;
delete p; // Free memory
```

- The memory remains allocated until explicitly `deleted` or the program exits
- (It is up to the OS to free on program exit; Linux and Windows do, some embedded OSs may not.)
- So, it is possible to leak memory by not freeing a pointer to it when you have finished with the memory.
- Copying pointers does not copy the associated memory.

Array allocation (heap)

- We can allocate memory for a block of variable size

```
int s = 100;
int* p = new int[s];
for(int i=0 ; i < s ; i++){
  p[i] = i;
}
delete[] p; // Free all 100 ints
```

Heap memory available outside fn

- The following is valid because memory allocated on the heap is not freed automatically:

```
int* allocateIntMem(const int a){
  int* ptr = new int[a];
  return ptr;
}

int* myData = allocateIntMem(10);
myData[0] = 6;
myData[9] = 8;
delete[] myData;
```

- `delete` frees the memory pointed to by `myData`, but `myData` still holds its original value.
- Any attempt to access memory through `myData` will be invalid hereafter, and (probably) lead to a seg-fault.

nullptr

- There is a C++-literal: `nullptr` usually used for an undefined pointer
- Trying to dereference this will lead to a segmentation-fault
- Usefully, `delete nullptr;` is valid, and does nothing
- This avoids the need for constructs such as

  ```
  if( p != nullptr ){
    delete p;
  }
  ```

  or equivalently

  ```
  if( p ){
   delete p;
  }
  ```

- Note also that short-cut evaluation means that expressions such as

  ```
  if( p && *p == 4 )
  ```

  are well-defined since `p` will not be dereferenced if it is `nullptr`

Pointer arithmetic

- Note that the square bracket notation is equivalent to dereferencing:
  `p[0]` and `*p` refer to the same thing.
- Arithmetic operations on a pointer are equivalent to `[]`:
  `p[1]` and `*(p+1)` are equivalent.
- Modifying a pointer is acceptable:

```
for(int i=0 ; i < 10 ; i++){
  *p = i;
  p = p+1;
}
```

- Adding 1 to a pointer advances it to point to the next item in memory of the same type.
- Note that this causes problems if you later try to `delete` a modified pointer. `delete` must be called on the original pointer returned by `new`.

Aliasing

- It is very easy to end up with aliasing effects again:

```
int* p = new int[N];
int* q = p;
q[10] = 10;
std::cout << p[10] << std::endl; // prints 10
```

- And multiple deletes on the same pointer are invalid:

```
delete[] p;
q[5] = 10; // Invalid
delete[] q; // Invalid
```

Smart pointers

- To ensure that allocated memory is always freed, consider a `shared_ptr`.
- This obeys scope, but can be copied, and the object pointed to is destroyed when the last pointer to it is deleted.

```
std::shared_ptr<double> p(new double);
*(p.get()) = 9.8;
std::shared_ptr<double> q = p;
std::cout << *(q.get()) << std::endl;
```

- and the memory is correctly freed once at the end of the program.
- A similar construct is `std::unique_ptr` which only allows a single pointer to a block of memory to exist.
- However, the above only supports `new[]` from C++17 onwards.

Failed allocation of memory

- If you are out of memory, or try to allocate too much memory:

  ```
  size_t bigMemory = 1L << 36;
  double* myBigData = new double[bigMemory];
  ```

  then an exception will be thrown.
- If you don't put any exception-handling code in, then your program will abort, which is probably what you want.
- `size_t` is an unsigned integer big enough to refer to all memory that might be needed on the current architecture.
- So, on a 64-bit system, `size_t` will be 8 bytes (64 bits) in size as the memory addressing uses 64 bits.
- (Note: `1L` is required to make `1L << 36` a 64-bit integer.)

Multi-dimensional arrays (heap)

- What about allocating multi-dimensional arrays on the heap?
- The short answer is that you can't
- The longer answer is that you need to create an `Array` class that deals with the access
- or allocate firstly a contiguous block of memory and then a set of pointers to successive rows of that.
- Either:

```
int *a = new int[N*M];
int aIJ = a[N*i + j];
```

- Or:

```
int *a = new int[N*M];
int **b = new int *[M];
b[0] = a; b[1] = a + N;
int aIJ = b[i][j];
```

sizeof

- If you need to determine how much memory is needed for an array, you can use `sizeof`:

```
size_t numBytesReqd = sizeof(double) * 100;
```

- This can also be applied to variables:

```
int b = 3;
size_t sizeB = sizeof(b); // e.g. 4 bytes
```

- Warning, the following does not behave as you might expect for an array:

```
double* a = new double[100];
size_t aSize = sizeof(a); // 8 bytes (on 64-bit machine)
```

  returning the size of a pointer variable.

- The `sizeof` operator returns a variable of type `size_t`, which is used whenever the size of an object in memory is required.

Outline

Function wrappers

- We may have the user choose an ODE solver function at run-time
- Assume we have two or more functions with a consistent interface:

```
double euler(double x, double t, double dt){
  return x + dt * y(x,t);
}
double rk2(double x, double t, double dt){
  // RK2 code here
  return x + dt * dy;
}
```

- and we want to call whichever one of these the user has chosen.

Function wrappers ctd

- We could have a switch statement every time we need to call one or the other:

  ```
  double nextX;
  switch(odeSolver){
    case ODEsolver::EULER:
      nextX = euler(x,t,dt);
      break;
    case ODEsolver::RK2:
      nextX = rk2(x,t,dt);
      break;
  }
  ```

- However, if this needs to occur multiple times, then we end up with multiple copies of this code, and what happens if we want to add `RK4`?

Function wrappers

- We would like to store a reference to the function:

```
#include <functional>
std::function<double(double, double, double)> odeSolverFn;
switch(odeSolver){
  case ODEsolver::EULER:
    odeSolverFn = &euler;
    break;
  case ODEsolver::RK2:
    odeSolverFn = rk2; // & not necessary
    break;
}
```

- This is used as:

```
nextX = odeSolverFn(x,t,dt);
```

Function wrapper cost

- Since the compiler no longer knows which function will be called at compile-time, this introduces an extra redirection call at machine-code level
- This has a very small computational cost.
- Unless you profile your real-world code and find it's expensive, don't worry about the extra cost.
- Note that function wrappers have replaced function *pointers* which were harder to get right, and less powerful.
- In older code, you may see the more confusing:

```
double (*odeSolverFn)(double, double, double) = &euler;
```