Pointers and Memory in C++

A beginner's guide - by Steven Gabarró

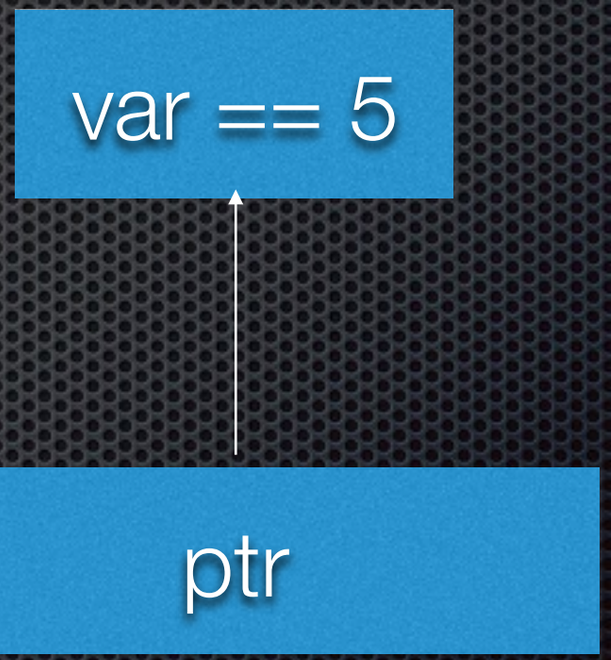
Slides and Examples…

- They can all be found at
  - https://github.com/bewchabbacc/pointers
- Note that all examples include “helpers.h” which contains a function printing the address and content of a variable

- The header also contains
  - ```
    #include <iostream>
    using namespace std;
    ```
  - Note that the “using” line has been placed in the header to make the examples shorter to read. It is usually a bad idea to include it in the global scope, and should be kept at the file level…
    ```

Virtual Memory - A refresher

- Each process has access to its own “virtual memory”, starting at address 0, mapped to physical memory by the OS
- Virtual Memory is typically organized as (this may vary slightly depending on architecture/OS):
  - OS/Architecture-Dependent Region (Off-limits)
  - Text (aka your compiled code)
  - Data (Initialized global and static data)
  - BSS (Block Started by Symbol - Uninitialized static/global data)
  - Heap (or Free memory, growing up)
  - Large empty space (Dynamic libraries are loaded in here)
  - Stack (growing down)
  - OS/Architecture-Dependent Region

The Heap

- The heap, or free store, is an area of memory that is shared throughout a process
- Data (variables / objects) stored in it can be read as long as the address of it is known
- Grows and shrinks by allocating / deallocating data (more on that later)

The Stack

- This is an area of memory that contains your stack frames (or Execution Block, or Activation Record)
- Each function call creates a new stack frame containing
  - The parameters sent to the function
  - The local variables of that function
  - Control data such as a reference to the previous stack frame and the return address
- It grows with each function call, and shrinks on each return

What is a pointer?

- Simply put: It is the address of something
- Pointers are typically the size of registers in your processor (4B in 32bit processors, 8B in 64bit)
- You can have pointers to:
  - Primitive type variables
  - Objects
  - Other pointers
    - Or pointers to pointers
    - Or pointers to pointers to pointers
    - etc…
  - Functions
  - Essentially, anything

Pointer Syntax

- To declare a pointer:
  - `type *var; // or type* var;`
- Ideally, initialize pointers with NULL if unsure about what to store in it
  - `type *var = NULL;`
- You can retrieve the address of any variable with the `&` symbol
  - ```
    int a = 42;
    int *pi = &a; // pi now holds the address of pi
    ```
- You can access the block pointed var by *dereferencing* it
  - `*var = 5;`
    ```

Visual Example

- `int var = 5;`
  `int *ptr = &var;`



Common Issues

- Do NOT assume pointers get initialized automatically with a 0, so you should ALWAYS give it a value (e.g. NULL (or nullptr in C++11))
- Understand the difference between
  - `type *a, b, c;` AND `type* a, b, c;`
  - In essence, NEVER do `type * a, b, c;`

Proper uses of * (other than multiplications)

- `type *symbol`
  - Declaration of a pointer (also used in function parameter declarations)
  - `int *pi = NULL;`
  - `void f(int *pi) { … }`
- `type*`
  - Used when declaring (but not implementing) methods
  - `void f(char*);`
- `*var`
  - Used to dereference a pointer (or reading what the pointer points to)
  - Note that the size read will depend on the type of the pointer!
  - `*pi = 5; // copies 5 into the integer block pointed by pi`
- `type (*name) (params…)`
  - Function pointers, try to avoid them unless absolutely necessary
  - `int (*operation)(int, int);`

Proper uses of &

- `&var`
  - Get the address of the variable var
- `type& var`
  - Used in function declarations to specify var is sent by reference
  - `void f(int& var) { … }`
- `var1 & var2`
  - Binary “and” operation between two variables
- `expr1 && expr2`
  - Logical “and” between two expressions

What NOT to do…

- `&&var`
  - This would mean get the address of the address of var
  - In reality it's nonsense!
    - Note: This is not the same as the && used in the C++11 "move" operator
- `*(&var)`
  - That would mean dereference the address of var
  - That's silly, just use var…
- `int *pi = (int*)5;`
  - While technically possible, this would mean that pi is now a pointer to the memory address 5!!! This would result in unpredictable code, and luckily most compilers will warn you about it
- Dereferencing a Bad pointer
- Returning the address of a Stack variable (compilers will complain about it)

```
Executor:pointers bewchy$ make test3
c++ -std=c++03 -Wall -pedantic-errors   -c -o test3.o test3.cpp
test3.cpp:8:11: warning: address of stack memory associated with local variable 'j' returned
      [-Wreturn-stack-address]
  return &j; // YIKES!! It hurts me to type this...
         ^
1 warning generated.
c++ -std=c++03 -Wall -pedantic-errors test3.o -o test3.out


#####  test3  #########
@: 0x7fff528deb00 Sz: 8 Val: 0x7fff528deac8
pi1 address 0x7fff528deac8 with data 2071065336
@: 0x7fff528deaf8 Sz: 8 Val: 0x7fff528deac8
pi2 holds 0x7fff528deac8 with data 2071065336
pi1 holds 0x7fff528deac8 with data 2071065336
```

```cpp
#include "helpers.h"

int *f(int i)
{
  int j = 5;

  j += i;
  return &j; // YIKES!! It hurts me to type this...
}

int main()
{
  int *pi1 = NULL;
  int *pi2 = NULL;

  pi1 = f(6);
  print_info(pi1);
  cout << "pi1 address " << pi1 << " with data " << *pi1 << endl;
  pi2 = f(5);
  print_info(pi2);
  cout << "pi2 holds " << pi2 << " with data " << *pi2 << endl;
  cout << "pi1 holds " << pi1 << " with data " << *pi1 << endl;


  return 0;
}
```

What is a bad pointer?

- Cristobal Colon statue in Barcelona (on the opening slide)
- He should be pointing to the Americas, but instead…

But really: What is a Bad Pointer?

- Pointer pointing to an area that you are not in control of
- Best case scenario, it's an area protected by the OS and you will get a segmentation fault when attempting to dereference the pointer
- Worst case scenario, you just played around with a block of memory that was holding something important needed elsewhere in your code (yikes!)

```cpp
#include "helpers.h"

void f_ptr(int *x)
{
  cout << "x is a pointer sent by value->";
  print_info(x);
}

void f_ptr_ref(int* &x)
{
  cout << "x is a pointer sent by reference->";
  print_info(x);
}
```

```
Executor:pointers bewchy$ g++ -DCRASHME test1.cpp -o test1.out
Executor:pointers bewchy$ ./test1.out
Segmentation fault: 11
Executor:pointers bewchy$
```

```cpp
int main()
{
  int i = 5;
  int *pi = &i;
  int *pi2 = NULL; // Dereferencing this will seg-fault!
  int *pi3; // THIS IS A BAD IDEA!

#ifdef CRASHME
  int *pi4 = (int*)5; // THIS IS EVEN WORSE!!!

  *pi4 = 10; // Congrats! you just got a segfault! I hope you're proud
#endif

  cout << "i->";
  print_info(i);
  cout << "pi->";
  print_info(pi);
  cout << "*pi->";
  print_info(*pi);
  i = 8;
  cout << "i->";
  print_info(i);
  cout << "pi->";
  print_info(pi);
  cout << "*pi->";
  print_info(*pi);

  cout << "\nLet's see our other pointers..." << endl;
  print_info(pi2);
  print_info(pi3);

  f_ptr(pi);
  f_ptr_ref(pi);
  return 0;
}
```

```
Executor:pointers bewchy$ make test1
c++ -std=c++03 -Wall -pedantic-errors   -c -o test1.o tes
c++ -std=c++03 -Wall -pedantic-errors test1.o -o test1.ou

#####  test1  #########
i->@: 0x7fff5b5b7b28 Sz: 4 Val: 5
pi->@: 0x7fff5b5b7b20 Sz: 8 Val: 0x7fff5b5b7b28
*pi->@: 0x7fff5b5b7b28 Sz: 4 Val: 5
i->@: 0x7fff5b5b7b28 Sz: 4 Val: 8
pi->@: 0x7fff5b5b7b20 Sz: 8 Val: 0x7fff5b5b7b28
*pi->@: 0x7fff5b5b7b28 Sz: 4 Val: 8

Let's see our other pointers...
@: 0x7fff5b5b7b18 Sz: 8 Val: 0x0
@: 0x7fff5b5b7b10 Sz: 8 Val: 0x0
x is a pointer sent by value->@: 0x7fff5b5b7ab8 Sz: 8 Val: 0x7fff5b5b7b28
x is a pointer sent by reference->@: 0x7fff5b5b7b20 Sz: 8 Val: 0x7fff5b5b7b28
```

Beware of pointer arithmetic!

- Note that basic operations may not behave as you expect
- `ptr = ptr2`
  - Copies the data in the pointer, which is an address!
  - Does not copy the data pointed by the pointers
- `ptr++ or ++ptr`
  - Increments the pointer by *sizeof(*ptr)* NOT just 1 (unless it is a char*)
  - e.g.
    - `int *ptr = &i; ++ptr;`
      - Now ptr is pointing sizeof(int) bytes AFTER i
- `*(ptr++) and (*ptr)++`
  - *(ptr++) will dereference the pointer ptr, then increment it by sizeof(*ptr)
  - (*ptr)++ will increment the data pointer by the pointer

The `new` and `delete` keywords

- While you can always get the address of something, you typically want the address of something that is NOT in the stack
- In order to put things in the Heap, we use *new*, to remove them we use *delete*
- `int *pi = new int;`
  - This creates a new block that can hold an integer in the Heap, and returns the address to pi
  - While similar to C's *malloc*, they are not interchangeable (especially when allocating objects, as *new* calls constructors but *malloc* doesn't)
- `delete pi;`
  - This marks the Heap block pointed by pi as *free*
  - This is similar to C's *free*, but not interchangeable (this will call object destructors)

EXTREMELY IMPORTANT!

DO NOT *free* SOMETHING ALLOCATED WITH *new*
DO NOT *delete* SOMETHING ALLOCATED WITH *malloc*
DO NOT *free* NOR *delete* THE SAME THING TWICE
DO NOT *free* NOR *delete* A STACK VARIABLE

```cpp
#include "helpers.h"

int *f()
{
  int *ret = new int;

  *ret = 42;
  return ret;
}

int main()
{
  int i = 5;
  int *pi = new int;
  int *pi2 = f();

  cout << "pi was allocated but not initialized" << endl;
  print_info(pi);
  print_info(*pi);
  *pi = i;
  print_info(pi);
  print_info(*pi);
  cout << "Let's checkout pi2: \n";
  print_info(pi2);
  print_info(*pi2);

  delete pi;
  delete pi2;
  return 0;
}
```

```
Executor:pointers bewchy$ make test2
c++ -std=c++03 -Wall -pedantic-errors   -c -o test2.o test2.cpp
c++ -std=c++03 -Wall -pedantic-errors test2.o -o test2.out


#####  test2  #########
pi was allocated but not initialized
@: 0x7fff5a19eb10 Sz: 8 Val: 0x7fddc8c04a20
@: 0x7fddc8c04a20 Sz: 4 Val: 0
@: 0x7fff5a19eb10 Sz: 8 Val: 0x7fddc8c04a20
@: 0x7fddc8c04a20 Sz: 4 Val: 5
Let's checkout pi2:
@: 0x7fff5a19eb08 Sz: 8 Val: 0x7fddc8c04a30
@: 0x7fddc8c04a30 Sz: 4 Val: 42
```

BEWARE!

- If you do not free a pointer and lose track of where it is, you just caused a *memory leak*
- This means you now have a reserved block of memory in the heap you have lost track of, taking up space for no reason
- e.g.
  - ```
    int i = 42;// makes a local block
    int *pi = new int; // allocates a heap block
    pi = &i; // overwrites the heap pointer with
    the address of the local i
    ```
    ```

Basic arrays

- Arrays are a contiguous block of memory holding data of a certain type
- The “name” of the array is in fact a pointer to the first byte of that block
- When you do *arr[i]* it is the same as doing
  - `*(arr + (i * sizeof(*arr)))`
  - or `(arr+(i*sizeof(arr[0])))[0]`
  - This means that it is technically legal to do *arr[-1]* but DO NOT DO IT!

delete[ ] and new[ ]

- In order to allocate a “raw” array of type T, you can do
  - `T arr[n]; // this puts the array in the stack`
  - `T arr[] = {data1, data2, data3, …};`
  - `T *arr2 = new T[n]; // this puts the array in the heap`
- Only put arrays in the heap if they may be needed outside the scope of your function
  - Better yet, use containers (see later on)
- To delete an array from the Heap, use delete []
  - `delete [] arr2;`
- Do NOT delete [ ] stack arrays
- Do NOT forget the [ ] when deleting arrays!

```cpp
#include "helpers.h"

int main()
{
  int arr[] = {1, 2, 3, 4, 5};
  int arr2[5];
  int *arr3 = new int[5];

  for (int i = 0; i < 5; ++i)
    {
      arr2[i] = 2*i;
      arr3[i] = 3*i;
    }
  cout << "Arr info: ";
  print_info(arr);
  cout << "\nArr2 info: ";
  print_info(arr2);
  cout << "\nArr3 info: ";
  print_info(arr3);
  cout << "\nArr[1]: ";
  print_info(arr[1]);
  cout << "\nArr2[2]: ";
  print_info(arr2[2]);
  cout << "\nArr3[3]: ";
  print_info(arr3[3]);
  cout << endl;

  delete [] arr3;
  return 0;
}
```

```
Executor:pointers bewchy$ make test4
c++ -std=c++03 -Wall -pedantic-errors   -c -o test4.o test4.cpp
c++ -std=c++03 -Wall -pedantic-errors test4.o -o test4.out


#####  test4  #########
Arr info: @: 0x7fff55831ab0 Sz: 20 Val: 0x7fff55831ab0

Arr2 info: @: 0x7fff55831a90 Sz: 20 Val: 0x7fff55831a90

Arr3 info: @: 0x7fff55831a70 Sz: 8 Val: 0x7fb04ac04a20

Arr[1]: @: 0x7fff55831ab4 Sz: 4 Val: 2

Arr2[2]: @: 0x7fff55831a98 Sz: 4 Val: 4

Arr3[3]: @: 0x7fb04ac04a2c Sz: 4 Val: 9
```

Allocating Objects: constructors and destructors

- When you create an object from a class, its constructor gets called
- When it is destroyed, the destructor is called
- If your object is in the Stack, constructor is called when you create it, the destructor is called when you “fall out of scope”
- If you want your object to be in the Heap, constructor/destructor will be called when you do new/delete
- To access data members and methods of a Stack object do
  - `obj.method()`
- To access data members and methods through a pointer to an object you could do
  - `(*obj).method() // UGLY!`
  - `obj->method() // NICER!`

```cpp
#include "helpers.h"
#include <stdexcept>

class IntArrWrapper
{
private:
  int cap;
  int *data;
public:
  IntArrWrapper() {cap = 5; data = new int[5];}
  IntArrWrapper(int c) {
    if (c < 5)  // we're forcing a minimum of 5 ints (why not)
      c = 5;
    cap = c;
    data = new int[c];
  }
  ~IntArrWrapper() { delete [] data; cout << "Deleted the data!" << endl; }
  int& operator[] (int n) throw (out_of_range) {
    if (n < 0 || n >= cap)
      throw out_of_range("Illegal index!\n");
    return data[n];
  }
  int length() const {
    return cap;
  }

  friend ostream& operator<< (ostream&, const IntArrWrapper&);
};

ostream& operator<< (ostream& o, const IntArrWrapper& a)
{
  o << "\nCap var at: " << &(a.cap);
  o <<"\ndata at " << &(a.data) << " its pointer is " << a.data << endl;
  return o;
}
```

```cpp
int main() {
  IntArrWrapper arr;
  IntArrWrapper *arr2 = new IntArrWrapper(10);

  cout << "ARR:\n";
  print_info(arr);
  cout << "ARR2:\n";
  print_info(arr2);
  cout << "*ARR2:\n";
  print_info(*arr2);
  arr[1] = 42;
  print_info(arr[1]);

  delete arr2;
  return 0;
}
```

```
#####  test5  #########
ARR:
@: 0x7fff5b927b28 Sz: 16 Val:
Cap var at: 0x7fff5b927b28
data at 0x7fff5b927b30 its pointer is 0x7fdd73500000

ARR2:
@: 0x7fff5b927b20 Sz: 8 Val: 0x7fdd73500020
*ARR2:
@: 0x7fdd73500020 Sz: 16 Val:
Cap var at: 0x7fdd73500020
data at 0x7fdd73500028 its pointer is 0x7fdd73500030

@: 0x7fdd73500004 Sz: 4 Val: 42
Deleted the data!
Deleted the data!
```

Inheritance and Object Allocation

- If you are using inheritance be aware of constructor and destructor call order
- The topmost constructor in the class hierarchy is called first, working its way down to the object you constructed
- Destructors are called in reverse order (from lowest in the hierarchy, working up through the ancestors)

```cpp
class Base
{
private:
  static unsigned int baseCount;
protected:
  vector<string> data;
public:
  virtual ~Base()
  {
    --baseCount;
    cout << "Base Destructor! EXTERMINATE!!" << endl;
  }
  Base()
  {
    ++baseCount;
    cout << "Base Constructor!" << endl;
  }
  Base(unsigned int n)
  {
    ++baseCount;
    cout << "Base non-default constructor!" << endl;
    data.reserve(n);
  }
  void add(string s)
  {
    data.push_back(s);
  }
  static unsigned int classCount() {
    return baseCount;
  }
  virtual void print_data()
  {
    for (vector<string>::iterator it = data.begin(); it != data.end(); ++it)
      {
        cout << *it << " ";
      }
    cout << "\nThis Base object is at location: " << this << endl;
  }
};
```

```cpp
class Child: public Base
{
private:
  static unsigned int childCount;
  unsigned int count;
public:
  ~Child() {
    --childCount;
    cout << "Child destructor!" << endl;
  }
  Child() {
    ++childCount;
    count = 0;
    cout << "Child default constructor!" << endl;
  }
  Child(unsigned int n): Base(n)
  {
    ++childCount;
    count = n;
    cout << "Child non-default constructor!" << endl;
  }
  void print_data()
  {
    cout << "There are " << count << " items: ";
    Base::print_data();
    cout << "This Child object is at location: " << this << endl;
  }
  static unsigned int classCount() {
    return childCount;
  }
};

unsigned int Base::baseCount = 0;
unsigned int Child::childCount = 0;
```

```cpp
int main(int argc, char **argv)
{
  if (argc < 2)
    cout << "Use: ./test3.out string1 string2 string3...";
  Child c(argc-1);
  Base *b = &c;
  Child *c2 = new Child(argc - 1);
  for (int i = 1; i < argc; i++)
    {
      c.add(string(argv[i]));
      c2->add(string(argv[i]));
    }
  cout << endl << "Showing contents through Child class" << endl;
  c.print_data();
  cout << endl << "Showing contents through Child pointer class" << endl;
  c2->print_data();
  cout << endl << "Showing contents through base class\n" << endl;
  b->print_data();

  cout << "\nThere are " << Base::classCount() << " base objects made and ";
  cout << Child::classCount() << " children ones." << endl;

  delete c2;
  return 0;
}
```

```
Executor:pointers bewchy$ make test6
c++ -std=c++03 -Wall -pedantic-errors   -c -o test6.o test6.cpp
c++ -std=c++03 -Wall -pedantic-errors test6.o -o test6.out

#####  test6  #########
Base non-default constructor!
Child non-default constructor!
Base non-default constructor!
Child non-default constructor!

Showing contents through Child class
There are 9 items: This is a call to all my past resignations
This Base object is at location: 0x7fff5d4b19a8
This Child object is at location: 0x7fff5d4b19a8

Showing contents through Child pointer class
There are 9 items: This is a call to all my past resignations
This Base object is at location: 0x7fada0404b20
This Child object is at location: 0x7fada0404b20

Showing contents through base class

There are 9 items: This is a call to all my past resignations
This Base object is at location: 0x7fff5d4b19a8
This Child object is at location: 0x7fff5d4b19a8

There are 2 base objects made and 2 children ones.
Child destructor!
Base Destructor! EXTERMINATE!!
Child destructor!
Base Destructor! EXTERMINATE!!
```

Naked pointers vs containers

- Until now we have talked about “naked pointers” aka “old school pointers”
- IF you chose to use naked pointers and allocate space yourself, remember that it is YOUR job to clean it up (collect your own garbage!)
- If you are trying to store data in memory, avoid using raw pointers / arrays. Use containers instead!
  - They handle memory themselves
  - Will grow/shrink as needed
  - Will get deleted for you (unless you allocated a pointer to a container, in which case you still need *delete*)
  - So PLEASE use things like *vector, deque, list, stack, queue, priority_queue, set, multiset, map,* or *multimap*

More Pointers Danger

- NEVER *delete* a stack variable / object / array
- NEVER *delete* something that is not a pointer
- NEVER assume pointers are initialized for you
- If you own a pointer, *delete* it when done with it! This includes pointers you did not allocate but were created for you by other functions
- Watch out for pointer arithmetic errors!
- If you pass a pointer to a function, by default you are sending a copy of that pointer
- If you have the choice between passing data by reference or passing a pointer to your data, send by reference!
- If you make a copy of a pointer, don't delete it twice! *delete* gets rid of the pointee, not the pointer!
- If you need data in the Heap, consider using a container or make a class that will take care of the data with a proper constructor and destructor

Quiz! Find the bugs!

- arr is a copy of the pointer ptr, so changes to arr don't affect ptr
- Adding one to a pointer will place your pointer 1B into the first number, rather than going to the next number in the array
- There's a memory leak caused by the allocation of ptr followed by the overwriting of the pointer with the data array
- The delete is wrong!
  - It has nothing to do with delete []
  - Since you overwrote ptr with data, now ptr refers to a stack array so DON'T DELETE IT!
- We need a return statement in main!

```
#include "helpers.h"

// Supposedly grabs the item pointer by arr, and
//   moves arr up by one item
int each(int *arr)
{
  int ret = *arr;

  arr += 1;
  return ret;
}

int main()
{
  int data[] = {32, 54, 33, 76, 4, 26, 76};
  int *ptr = new int[7];

  ptr = data;
  for (int i = 0; i < 7; ++i)
    cout << each(ptr) << " ";
  delete ptr;
}
```

A peek into the “future”

- To avoid many of the problems you can get from using raw pointers, C++11 introduced “smart pointers”
  - `unique_ptr<type>`
    - Prevents the pointer from being copied
  - `shared_ptr<type>`
    - Only gets deleted with ALL copies of the pointer are no longer needed/used
  - `weak_ptr<type>`
    - Created as a copy of a shared_ptr that won’t affect the other shared_ptr, but will get erased when the shared_ptr it came from gets deleted
- Note that C++03 has *auto_ptr*
  - This was removed in C++11 and in order to future-proof your code, you should avoid using it!

Questions?