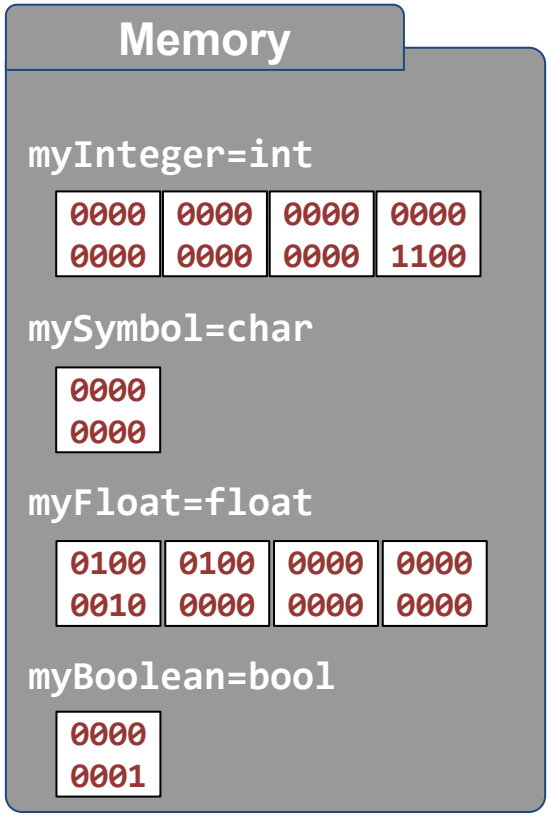
7. Pointers and Memory Allocation

7.1. Pointers

Reminder: Variables and objects reside in memory. Through the variable name, you can read and change the variable's value. Its type tells the compiler how.

```
/* reserving variables */
int main() {
  int myInteger = 12;   // store an integer
  char mySymbol;   // store one character
  float myFloat = 12.0f;  // store floating point
  bool myBoolean = true;  // store a boolean
  return 0;
}
```

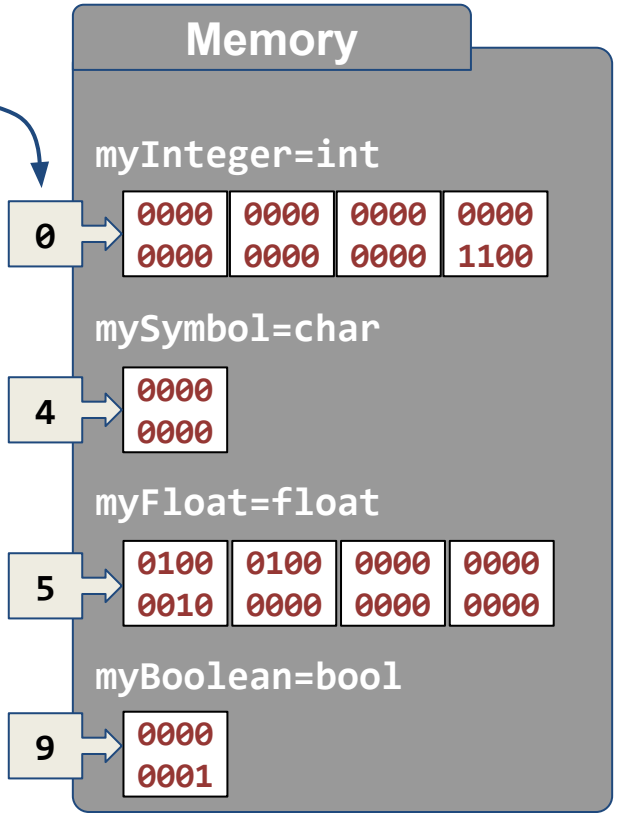


7.1. Pointers

Each memory location has a **memory address**

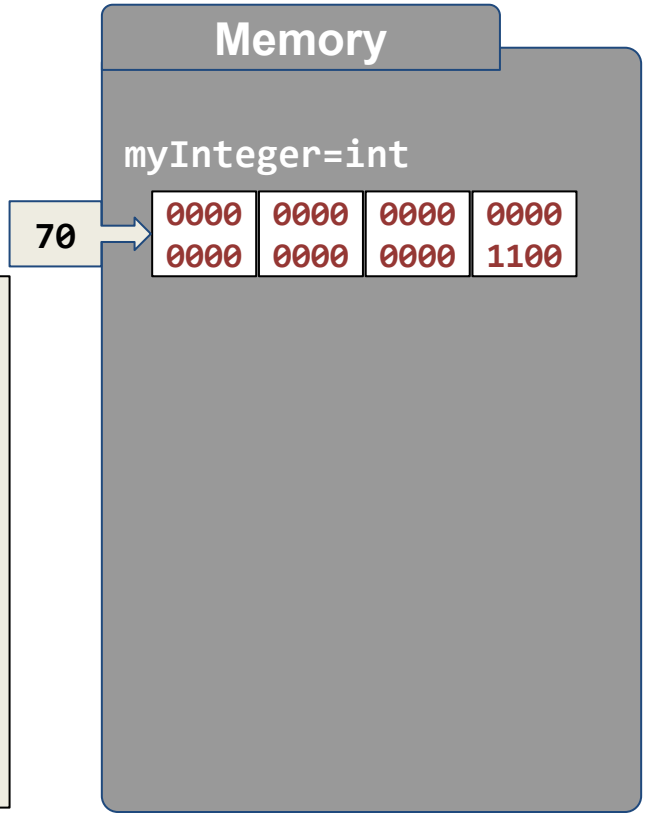
We assume here that one memory block occupies one byte.

```
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  char mySymbol;    // store one character
  float myFloat = 12.0f;  // store floating point
  bool myBoolean = true;  // store a boolean
  return 0;
}
```



7.1. Pointers

A pointer stores a **memory address and its associated type**

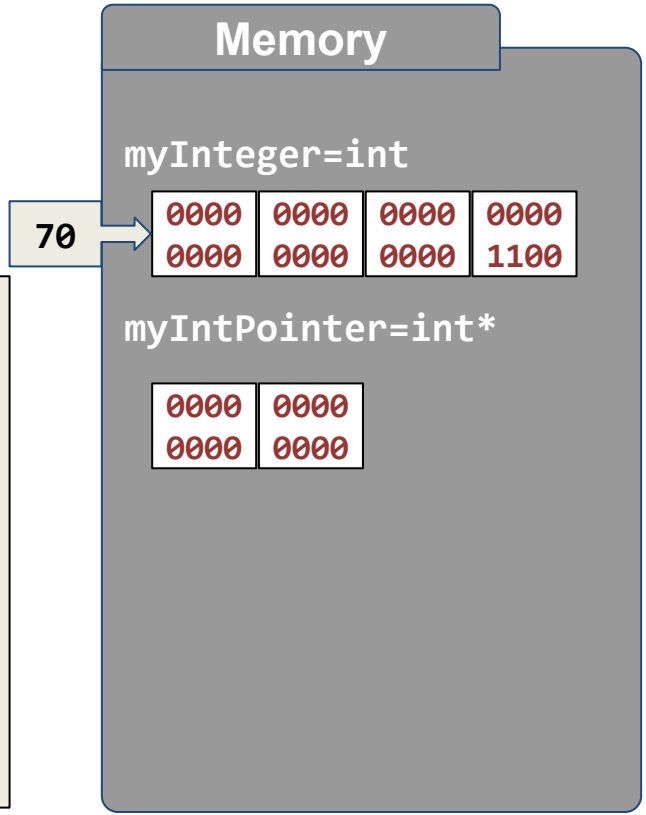



```c
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int *myIntPointer;    // store a pointer to int
  myIntPointer = &myInteger;  // store floating
  *myIntPointer = 17; // myInteger is now also 17
  return 0;
}
```

7.1. Pointers

A pointer stores a **memory address and its associated type.** Pointer variables are declared by using the * operator.

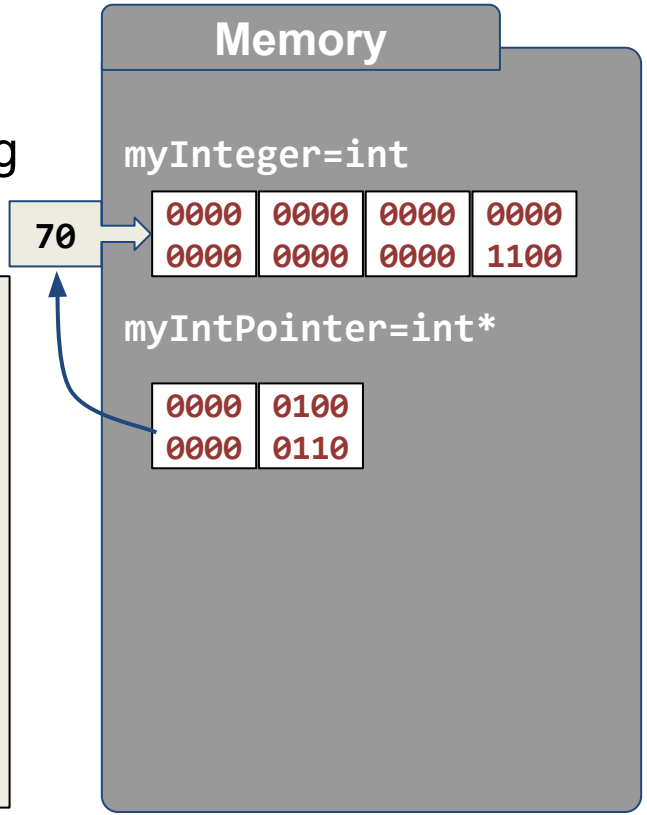
```
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int *myIntPointer;     // store a pointer to int
  myIntPointer = &myInteger;  // store floating
  *myIntPointer = 17; // myInteger is now also 17
  return 0;
}
```



7.1. Pointers

Pointer variables can obtain the address of existing variables of that type using the **&** operator (signifying the address of a variable)

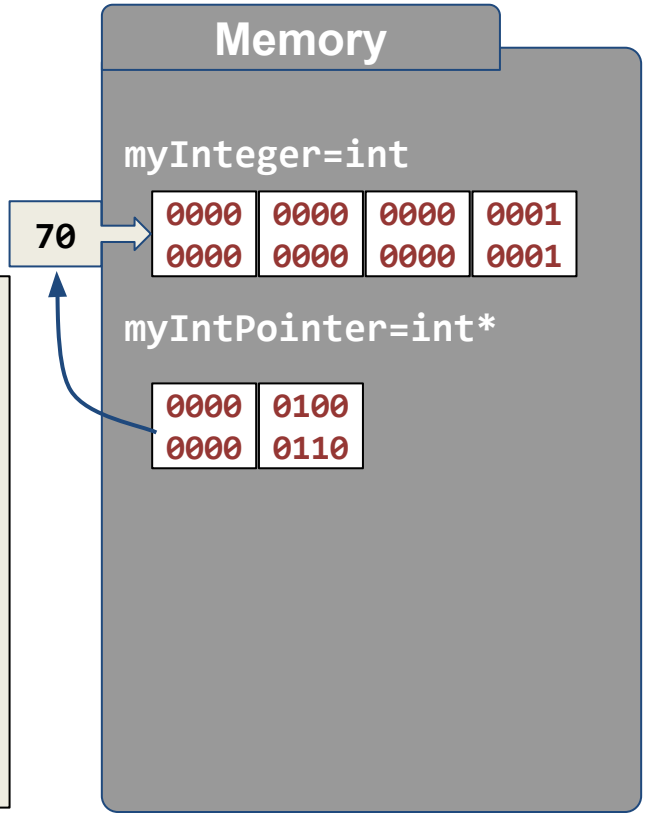
```
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int *myIntPointer;    // store a pointer to int
  myIntPointer = &myInteger;  // store floating
  *myIntPointer = 17; // myInteger is now also 17
  return 0;
}
```



7.1. Pointers

Dereferencing a pointer means following the pointer's content (memory address) and accessing the variable of that type

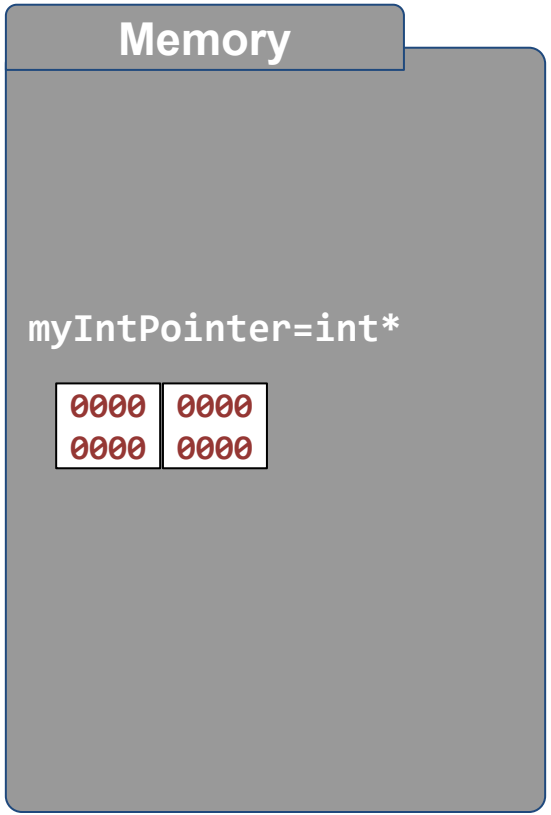
```
/* reserving variables */
int main() {
  int myInteger = 12;    // store an integer
  int *myIntPointer;    // store a pointer to int
  myIntPointer = &myInteger;  // store floating
  *myIntPointer = 17; // myInteger is now also 17
  return 0;
}
```



7.2. `new` and `delete`

A pointer assumes a memory address is already reserved for a variable. Indicate that the pointer isn't pointing to a valid variable with the `NULL` keyword:

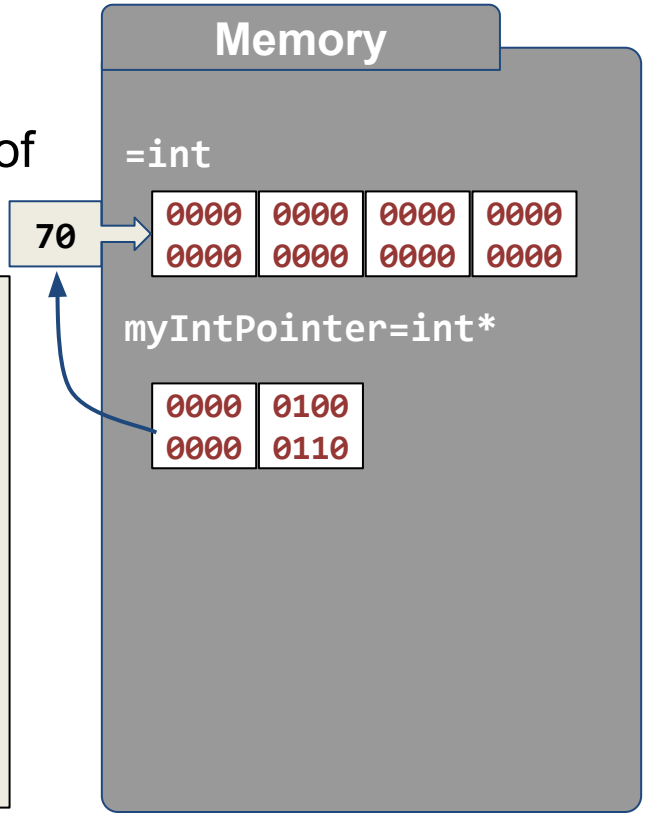
```
/* reserving variables through a pointer */
int main() {
  int *myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove the pointer
  return 0;
}
```



7.2. `new` and `delete`

A pointer assumes a memory address is already reserved for a variable. We can create the variable of the correct type through the `new` keyword:

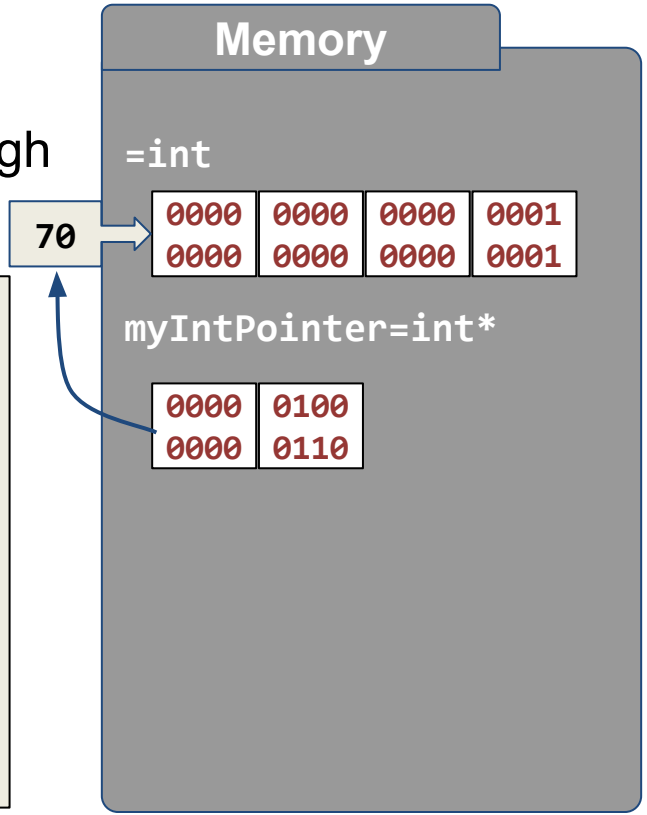
```
/* reserving variables through a pointer */
int main() {
  int *myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove the pointer
  return 0;
}
```



7.2. `new` and `delete`

A pointer assumes a memory address is already reserved for a variable. This variable can only through the pointer be read and changed:

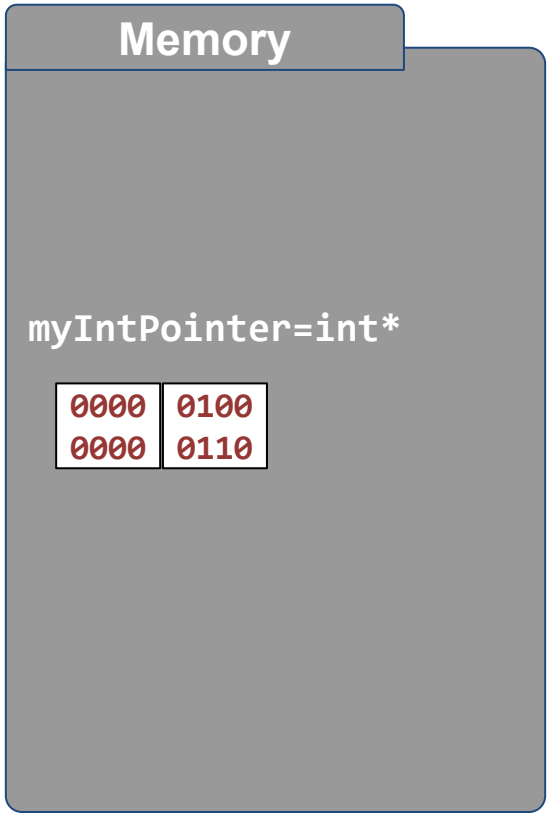
```
/* reserving variables through a pointer */
int main() {
  int *myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove the pointer
  return 0;
}
```



7.2. `new` and `delete`

A pointer assumes a memory address is already reserved for a variable. This variable can only through the pointer be read and changed:

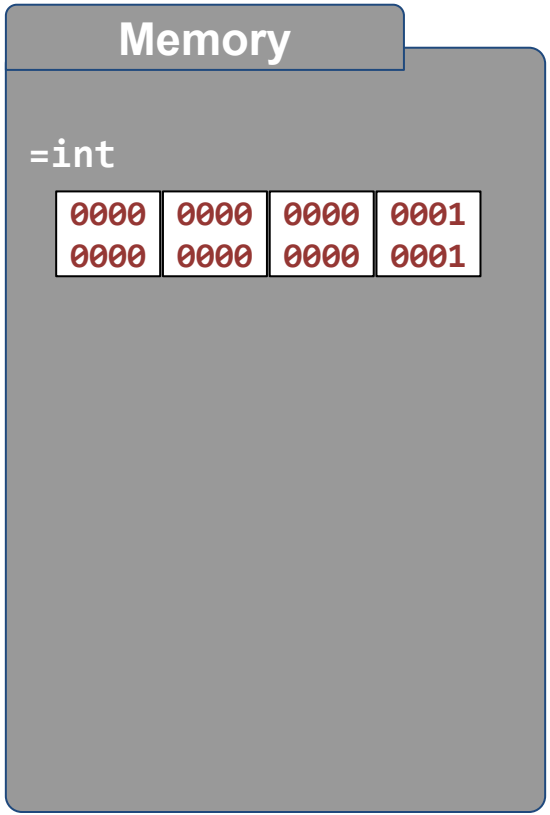
```
/* reserving variables through a pointer */
int main() {
  int *myIntPointer = NULL;  // pointer to int
  myIntPointer = new int;  // create the int
  *myIntPointer = 17;  // the int now holds 17
  delete myIntPointer;  // remove pointer's int
  return 0;
}
```



7.2. `new` and `delete`

Forgetting to `delete` a reserved variable causes a memory leak, since the pointer is removed and the variable cannot be reached anymore (but is reserved).

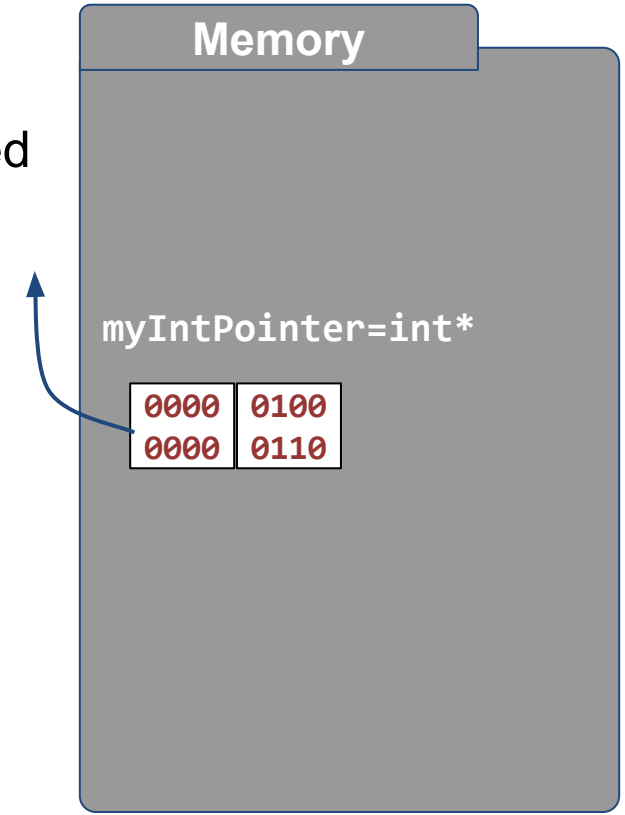
```
void myFunction() {
  int *myIntPointer = new int; // create int
  *myIntPointer = 17;
}
int main() {
  myFunction();
  // after the above function ends, myIntPointer
  // is removed, but not the int variable
  return 0;
}
```



7.2. `new` and `delete`

It is good practice to assign the pointer to `NULL` after `delete` (since the pointer still points to a non-reserved memory location, this is called a *dangling pointer*).

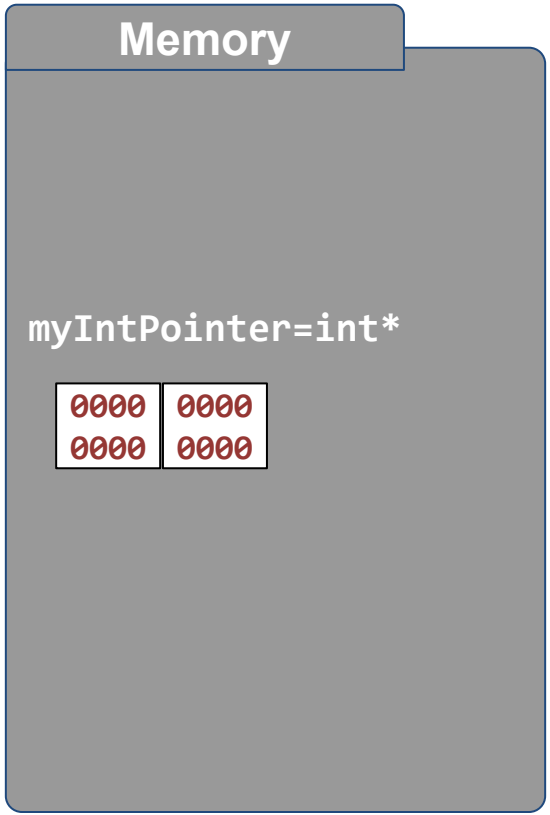
```
void myFunction() {
  int *myIntPointer = new int; // create int
  *myIntPointer = 17;
  delete myIntPointer;  // remove pointer's int
  myIntPointer = NULL;  // point to NULL
}
int main() {
  myFunction();
  return 0;
}
```



7.2. `new` and `delete`

It is good practice to assign the pointer to `NULL` after `delete` (since the pointer still points to a non-reserved memory location, this is called a *dangling pointer*).

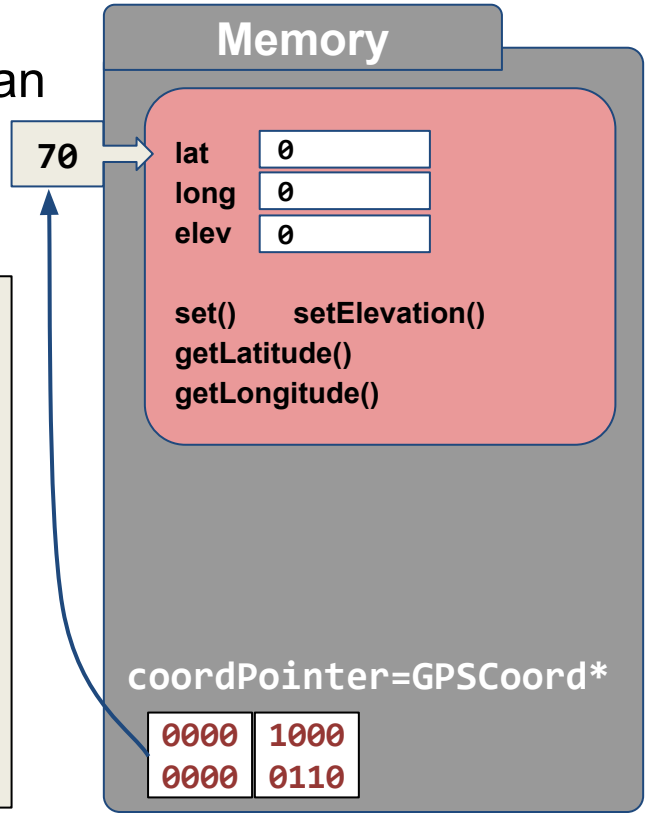
```
void myFunction() {
  int *myIntPointer = new int; // create int
  *myIntPointer = 17;
  delete myIntPointer;  // remove pointer's int
  myIntPointer = NULL;  // point to NULL
}
int main() {
  myFunction();
  return 0;
}
```



7.3. Creating and deleting objects

Pointers can also point to classes' objects. These can similarly be created and deleted in memory, too.

```
void myFunction() {
  GPSCoord *coordPointer = new GPSCoord();
  coordPointer->set(0, 0);  // access method
  delete coordPointer;  // remove object
  coordPointer = NULL; // avoid dangling pointer
}
int main() {
  myFunction();
  return 0;
}
```

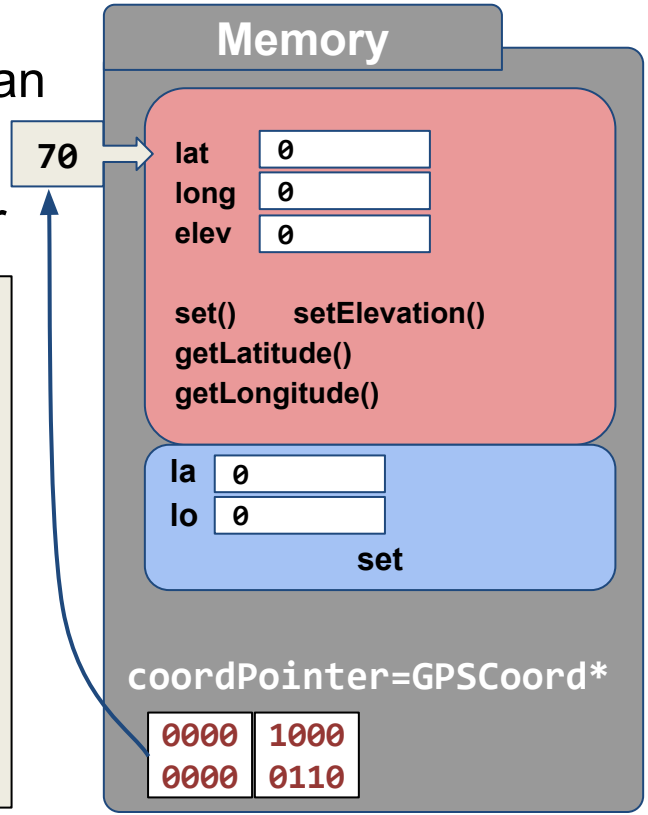


7.3. Creating and deleting objects

Pointers can also point to classes' objects. These can similarly be created and deleted in memory, too.

Attributes & methods are accessed with `->` operator

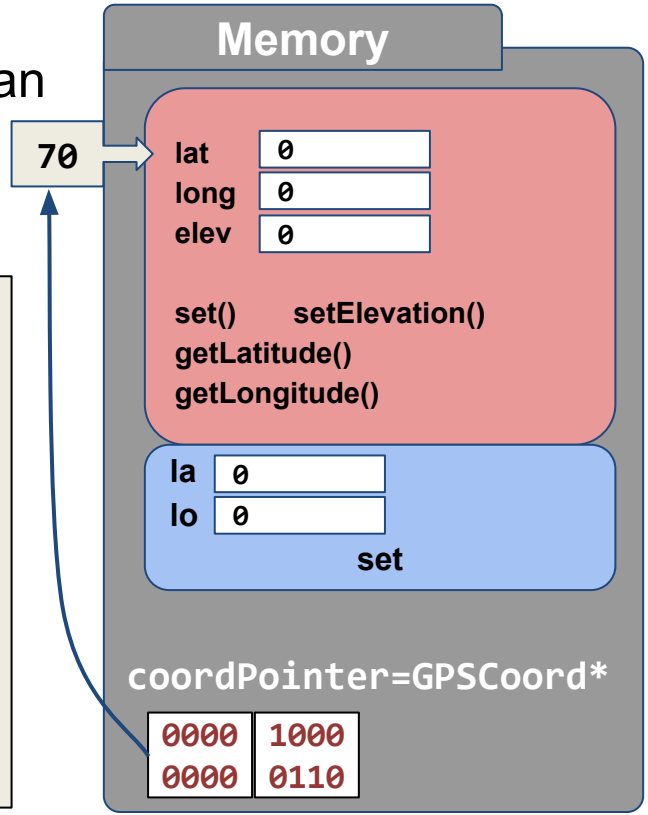
```
void myFunction() {
  GPSCoord *coordPointer = new GPSCoord();
  coordPointer->set(0, 0);  // access method
  delete coordPointer;  // remove object
  coordPointer = NULL; // avoid dangling pointer
}
int main() {
  myFunction();
  return 0;
}
```



7.3. Creating and deleting objects

Pointers can also point to classes' objects. These can similarly be created and deleted in memory, too.

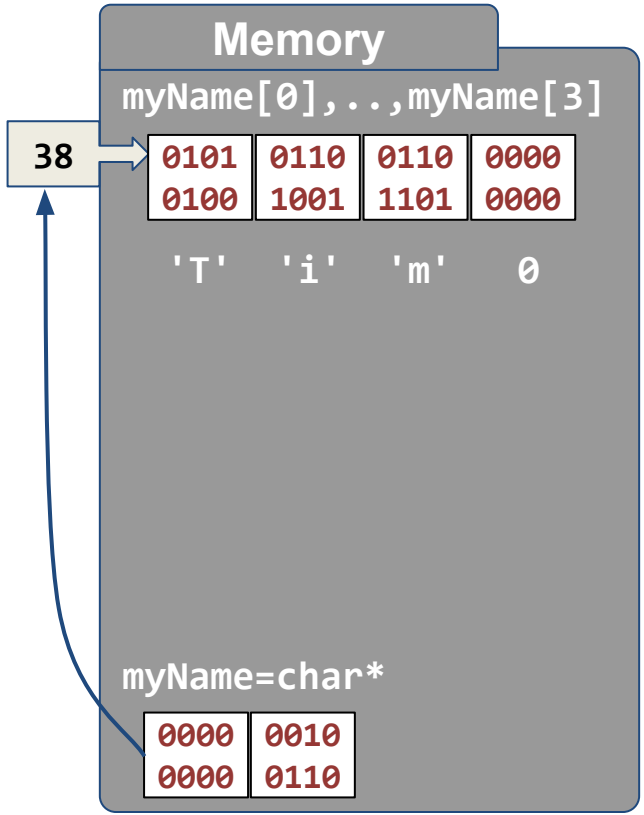
```
void myFunction() {
  GPSCoord *coordPointer = new GPSCoord();
  coordPointer->set(0, 0);  // access method
  delete coordPointer;  // remove object
  coordPointer = NULL; // avoid dangling pointer
}
int main() {
  myFunction();
  return 0;
}
```



7.4. Pointers and arrays

In C++, an array name is analogue to a pointer to the first element of the array:

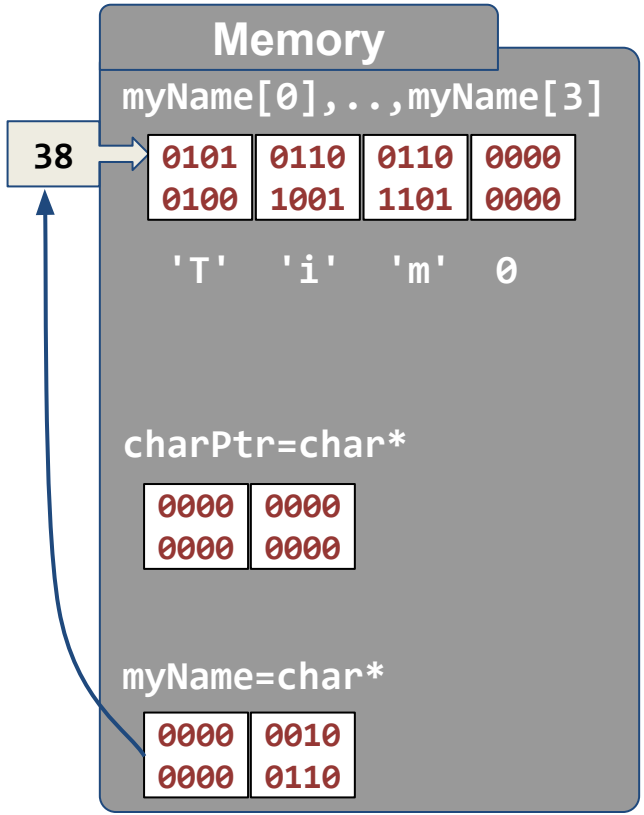
```
char myName[4] = "Tim";
char *charPtr = NULL;
charPtr = myName;  // myName == &myName[0]
// note that this is invalid: myName = charPtr;

std::cout << "1st: " << *(charPtr) << std::endl;
// → gives out 'T'
std::cout << "2nd: " << myName[1] << std::endl;
// → gives out 'i'
std::cout << "3rd: " << *(charPtr+2) << std::endl;
// → gives out 'm'
```



7.4. Pointers and arrays

In C++, an array name is analogue to a pointer to the first element of the array:

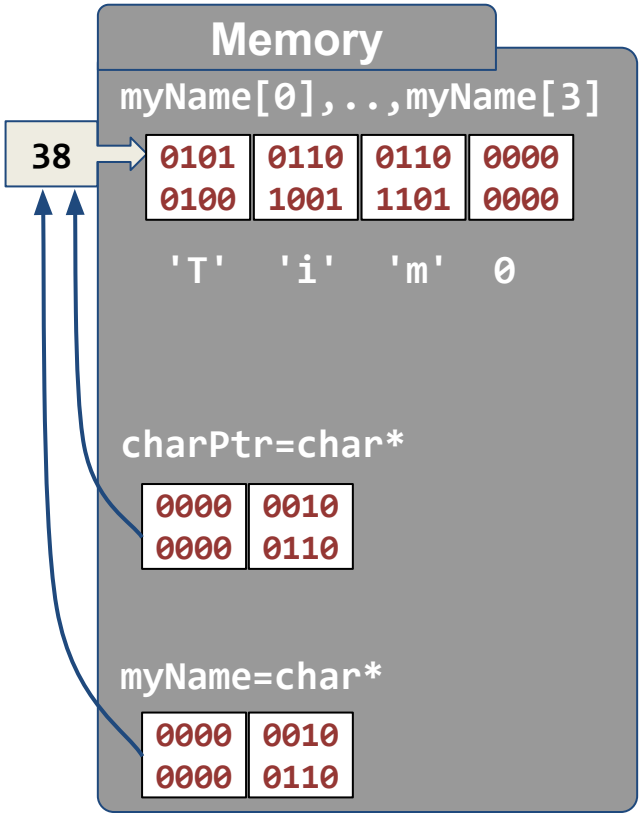
```
char myName[4] = "Tim";
char *charPtr = NULL;
charPtr = myName;  // myName == &myName[0]
// note that this is invalid: myName = charPtr;

std::cout << "1st: " << *(charPtr) << std::endl;
// → gives out 'T'
std::cout << "2nd: " << myName[1] << std::endl;
// → gives out 'i'
std::cout << "3rd: " << *(charPtr+2) << std::endl;
// → gives out 'm'
```



7.4. Pointers and arrays

In C++, an array name is analogue to a pointer to the first element of the array:

```
char myName[4] = "Tim";
char *charPtr = NULL;
charPtr = myName;  // myName == &myName[0]
// note that this is invalid: myName = charPtr;

std::cout << "1st: " << *(charPtr) << std::endl;
// → gives out 'T'
std::cout << "2nd: " << myName[1] << std::endl;
// → gives out 'i'
std::cout << "3rd: " << *(charPtr+2) << std::endl;
// → gives out 'm'
```



7.4. Pointers and arrays

Arrays can be dynamically allocated at run time with pointers:

```
// we receive a size variable here that we need an array around,
// but do not know how large it is at design time:
int size = fileData.getSize();

// We CAN create an array of the required size:
GPSCoord *myRoute = new GPSCoord[size];  // a route is created as points
for (int i = 0; i < size; i++) {
  fileData.readNext();  // read data from file, set these as route points:
  myRoute[i].set( fileData[0], fileData[1] );
  myRoute[i].setElevation( fileData[2] );
}

delete[] myRoute;  // for dynamically created arrays, delete needs []
myRoute = NULL;
```

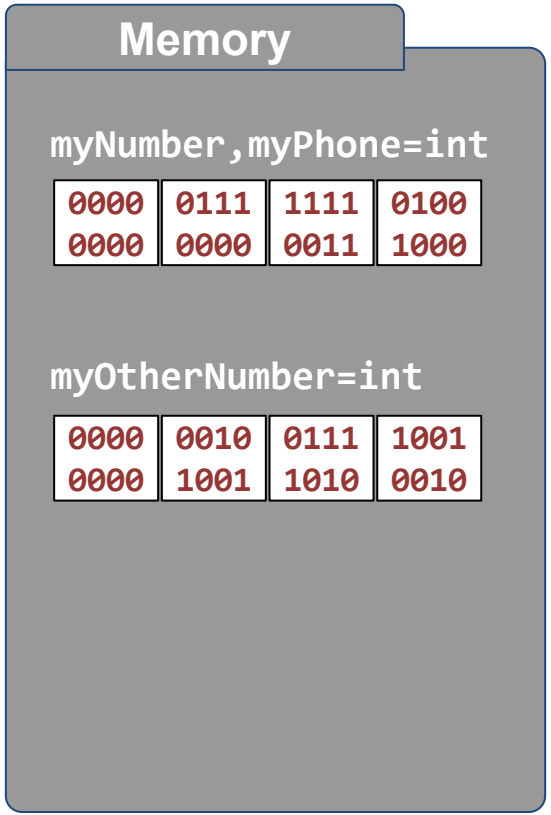
7.4. Pointers and arrays: Example 00 (difficulty level: 🌶️)

```
/* Create an array for which the length is given at runtime through an argument
   of the executable. The main function in C++ can also have two parameters:
   argc: integer containing the count of arguments in argv
   argv: array of strings holding command-line arguments.
   argv[0] is the command itself, argv[1] is first argument */
#include <iostream>

int main(int argc, char *argv[]) { // executable's arguments are passed
  // if an argument given, assume it is a number and convert from string:
  if (argc > 1) {
    int size = std::stoi(argv[1]);

    // add code to create an array of length size, and fill it with increasing
    // numbers from 1 till size, display these, and then delete the array

  }
  return 0;
}
```

7.5. References

In C++, a reference is like an alias, or second name, for a variable. A reference can only be initialized, but never reassigned to another variable.

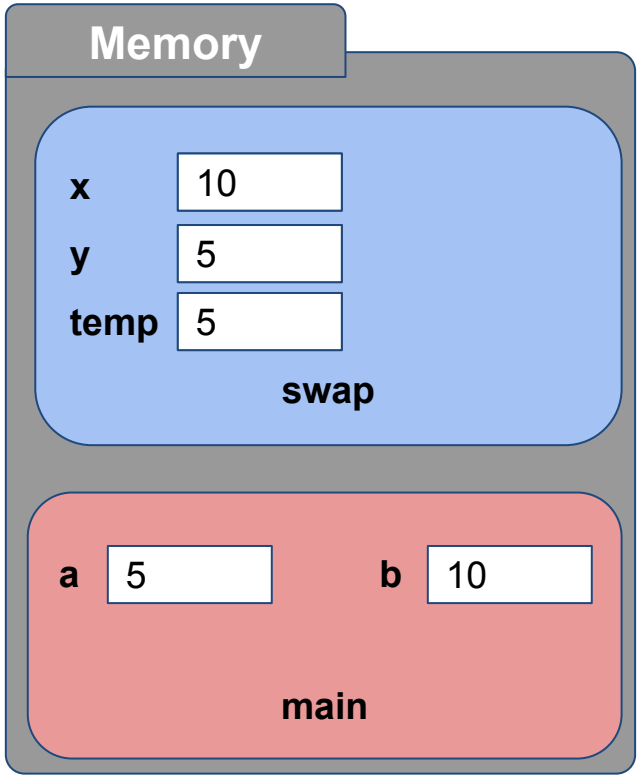
```
int myNumber = 7402312;
int &myPhone = myNumber;  // & in this declaration
// shows that this is a reference. From here on,
// myNumber and myPhone name the same variable.
int myOtherNumber = 2718354;
myPhone = myOtherNumber;
// → Now all three variable names myNumber, myPhone,
// and myOthernumber, have the same value: 2718354
// &myPhone = myOtherNumber; is invalid
```



7.6. Call-by-Reference

Reminder: the swap function below is not going to work, because variables are **passed-by-value**

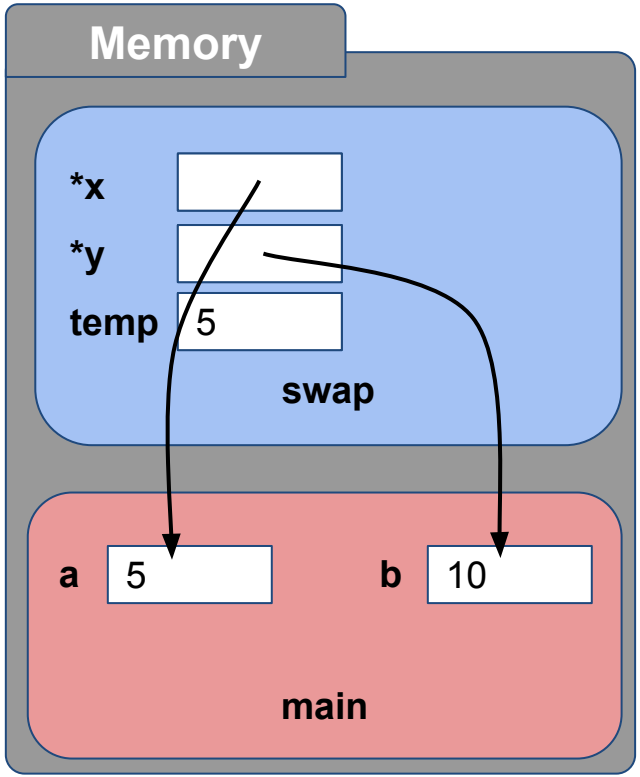
```
#include <iostream>  // output to terminal
void swap(int x, int y){
  int temp = 0;
  temp = x; x = y; y = temp;
}
int main()  {
  int a = 5, b = 10;
  swap(a, b);
  std::cout << a << ", " << b << std::endl;
  return 0;
}
```



7.6. Call-by-Reference

In C++, parameters can also be pointers. These are passed *by reference*, allowing swap to work:

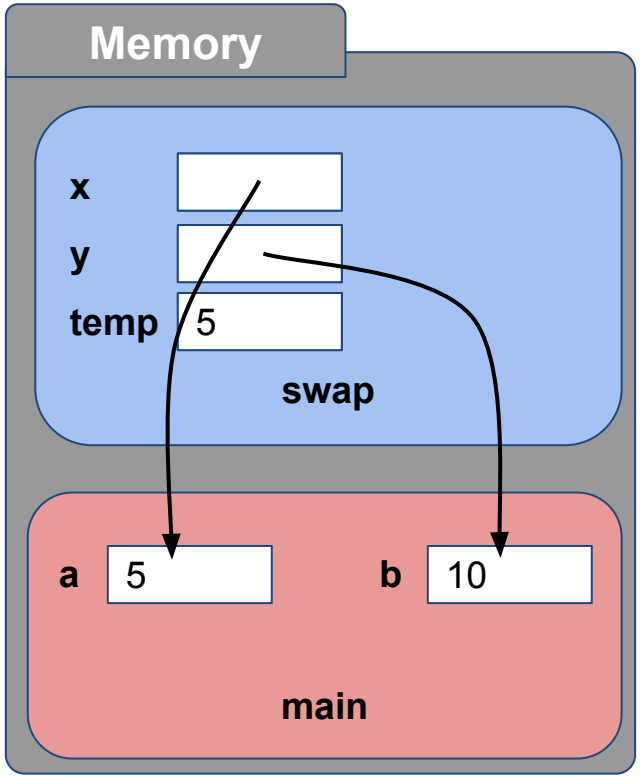
```
#include <iostream>  // output to terminal
void swap(int *x, int *y)  {
  int temp = 0;
  temp = *x; *x = *y; *y = temp;
}
int main()  {
  int a = 5, b = 10;
  swap(&a, &b);
  std::cout << a << ", " << b << std::endl;
  return 0;
}
```



7.6. Call-by-Reference

References have the same effect: These allow swap to work, too, and are safer and elegant:

```
#include <iostream>  // output to terminal
void swap(int &x, int &y)  {
  int temp = 0;
  temp = x; x = y; y = temp;
}
int main()  {
  int a = 5, b = 10;
  swap(a, b);
  std::cout << a << ", " << b << std::endl;
  return 0;
}
```



7.6. Call-by-Reference

When variables do not change, they can be passed as **`const`** references.

This is generally a clearer signature for developers calling our function/method:

```
#include <iostream>  // output to terminal
[[nodiscard("Please handle this function's return value.")]]
int sum3(const uint16_t &x, const uint16_t &y, const uint16_t &z) {
  return x + y + z;
}

int main() {
  auto a = 5, b = 10, c = 15;
  std::cout << sum3(a, b, c) << std::endl;
  return 0;
}
```

7.6. Call-by-Reference

Example 01 (difficulty level: 🌶️)

```
/* Write a class, called Check, below to illustrate in the main function that
   the pointer to an object of class Check b is indeed pointing to the object of
   class Check a. */
#include <iostream>

// write the class Check here

int main(int argc, char *argv[]) {
  Check a;  // a is an object of class Check
  Check *b = &a;  // assign address of a to pointer b to object of class Check
  if ( b->isThisMe( &a ) ) {
    std::cout << "&a is b \n";
  }
  return 0;
}
```

7.7. Copy Constructors

We know that functions and methods create a copy of actual parameters (*pass by value*), so how are objects dealt with? How do we copy objects?

```
void UTMCoord::from(GPSCoord coord) {
  // transforms from latitude-longitude to UTM coordinates
}

int main()  {
  GPSCoord place(50.88385, 8.02096);
  UTMCoord place2;
  place2.from( place );  // place's values are copied and passed
  return 0;
}
```

7.7. Copy Constructors

How do we copy objects? ⇒ With **copy constructors**

GPSCoord.h

```
class GPSCoord {  // GPS coordinate class
 public:
  GPSCoord() {;}  // default constructor
  GPSCoord(GPSCoord const &source);  // copy constructor
  void set(double lat, double lng);  // set latitude, longitude
  void setElevation(double elv);     // set elevation
  void print();  // output coordinates to console
 private:
  double lat, lng, elv;  // latitude, longitude, elevation
};
```

in GPSCoord.cpp

```
GPSCoord::GPSCoord(GPSCoord const &source) {
  lat = source.lat; lng = source.lng; elev = source.elv;
}
```

7.7. Copy Constructors

A copy constructor makes an object from another object of the same class, so in our example, we "clone" our GPSCoord object.

The copy constructor is implicitly called:

- when an object is **passed** to a function or method by value, or
- when a function or method **returns** an object

If a class does not implement a copy constructor, the C++ compiler will provide a default copy constructor, performing a ***member-wise copy*** (also known as ***shallow copy***)

So why do we ever have to implement a copy constructor ourselves?

7.7. Copy Constructors

An example of deep versus shallow copy:

```
class GPSTrace {  // class for a GPS trace

 public:
  GPSTrace(uint16_t numPoints);
  ~GPSTrace();
  // add a new point to trace at position pos:
  void setPoint(GPSCoord newPoint, uint16_t pos);
  [[nodiscard]] int print();  // print trace, forces return handling

 private:
  GPSCoord *points;  // pointer to GPS coordinates
  uint16_t numPoints;
};
```

7.7. Copy Constructors

An example of deep versus shallow copy:

```
GPSTrace::GPSTrace(uint16_t numpoints): numPoints(numpoints) {
  points = new GPSCoord[numPoints];
}

GPSTrace::~GPSTrace() {
  delete[] points; points = NULL; numPoints = 0;
}

void GPSTrace::setPoint(GPSCoord newPoint, uint16_t pos) {
  if (pos < numPoints) points[pos] = newPoint;
}
int GPSTrace::print() {  // output trace to console
  for (auto i = 0; i < numPoints; i++) points[i].print();
  return 0;
}
```

7.7. Copy Constructors

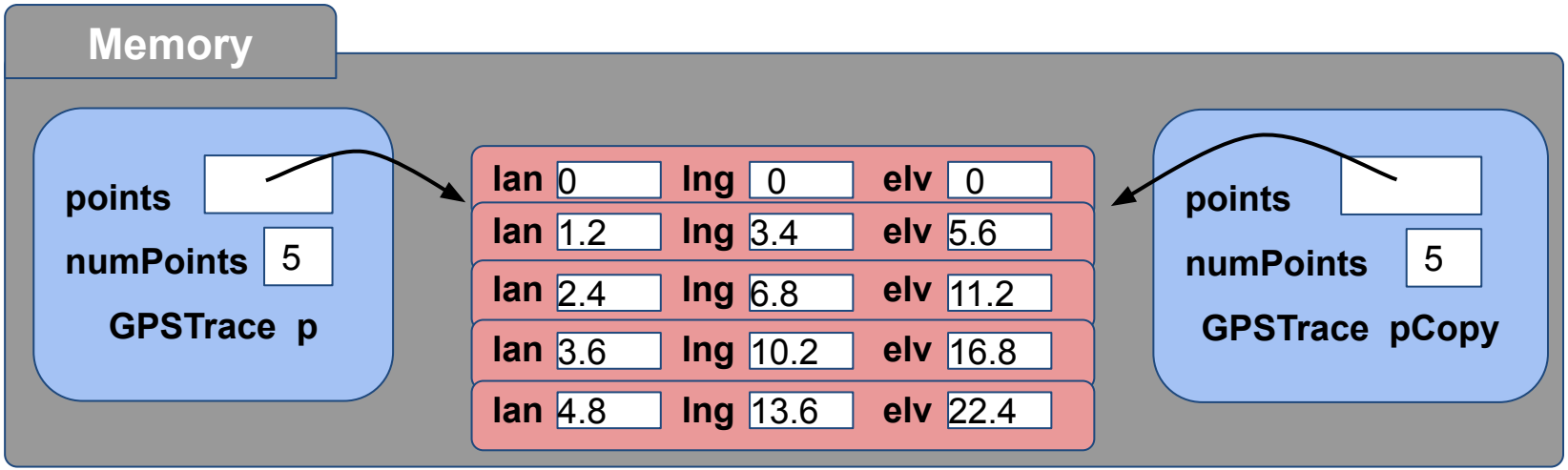
Example 02 (difficulty level: 🌶🌶)

```
/** An exercise illustrating shallow and deep copy: Add to the code of GPSTrace
    the necessary functionality that allows copying a GPSTrace and illustrate this
    in the main function below. */
#include <iostream>     // terminal output

// Classes GPSCoord and GPSTrace come here

int main() {
  GPSTrace t(5);
  for (auto i = 0; i < 5; i++ ) {  // fill in the GPS points
    GPSCoord point;
    point.set( i*1.2, i*3.4 ); point.setElevation( i*5.6 );
    t.setPoint( point, i );
  }
  return t.print();
}
```

7.7. Copy Constructors

Example 02 (difficulty level: 🌶🌶)

The standard (shallow) copy of **`p`** (e.g., **`pCopy`**) will result in this:



So the (deep) copy needs to be implemented in the copy constructor

7.8. `const` and `const` Pointers

Reminder: constants are defined with the **`const`** keyword, e.g.:

```
const int gpsTraceLength = 150;  // an integer constant
```

A constant can only be initialized, and a new value can not be assigned to a constant once it is defined. Usually this is done to protect the constant from being changed later on when this isn't planned.

7.8. `const` and `const` Pointers

`const` pointers can come in various forms, what matters is that everything on the left of the **`const`** keyword is constant. If **`const`** is on the full left, what is on its right is constant. **`const`** pointers need to be directly initialized:

```
int myInteger = 71;
const int constInt = 17;
int const *pointToConstInt = &constInt; // pointer to const int
int *const constPointToInt = &myInteger; // const pointer to int
int const *const cPointToCInt = &constInt; // const pointer to const int
const int *pointToConstInt2 = &constInt; // pointer to const int
```

7.8. `const` and `const` Pointers

`const` pointers protect pointers from being changed later in the code.

This means that changes such as: `*pointToConstInt = myInteger;`, `constPointToInt = &myInteger;`, `*cPointToCInt = myInteger;`, and `cPointToCInt = &myInteger;` will all result in an error.

For example: The **`this`** pointer in a class' methods is a **`const`** pointer to an object of the class, so cannot be changed to point to anywhere else but the current object.

7.8. `const` and `const` Pointers

`const` member functions or methods (aka *const qualified*):

```
int GPSTrace::print() const;  // output trace to console
```

`const` in them method's declaration *and* definition tells the compiler that the method should not modify the object (i.e., change the attributes)

- The compiler enforces this, and reports an error once the method's code tries to change its attributes
- Using const methods whenever possible will add guarantee that it will be used properly in the future

7.8. `const` and `const` Pointers

Example 03 (difficulty level: 🌶🌶🌶):

```
/** Print a mouse in the console, using a const pointer to avoid changes */
#include <iostream>     // terminal output
[[nodiscard]] auto *getBitmapAddress() {
    static char bitmap[] = "(^._.^)~";  // "bitmap" created in static memory
    return bitmap;  // return pointer to first element
}
int main() {
  // using a pointer to bitmap, and incrementing it, is possible:
  auto *mousePointer = getBitmapAddress();
  while ( *mousePointer != 0 )  std::cout << *(mousePointer++);
  std::cout << "\n";
  // Here mousePointer has changed, it's hard to get the original pointer.
  // Modify the above by protecting the pointer with const and redo the loop.
  return 0;
}
```

7.9. Passing functions to functions: Passing pointer to function

In C, functions can be passed as a parameter, which will be as a pointer to the function and is just the function's name:

```
#include <iostream>     // terminal output

int addTwo(int x) { return x + 2; }     // functions we can pass in callFunct
int timesFour(int x) { return x * 4; }  // since they match the signature

// callFunction takes a pointer to a function:
int callFunct(int x, int (*funct)(int) ) { return funct(x); /* = (*funct)(x) */ }

int main() {
  std::cout << "addTwo(149) = " << callFunct(149, addTwo) << "\n";
  std::cout << "timesFour(4) = " << callFunct(4, timesFour) << "\n";
  return 0;
}
```

7.9. Passing functions to functions: The `std::function`

In C++ 11 and onward, **`std::function`** can be used from **`<functional>`** (using templates, see later):

```
#include <iostream>     // terminal output
#include <functional>     // use std::function to pass functions as parameter

int addTwo(int x) { return x + 2; }     // functions we can pass in callFunct
int timesFour(int x) { return x * 4; }  // since they match the signature

// callFunction takes a pointer to a function:
int callFunct(int x, std::function<int(int)> func ) { return funct(x); }

int main() {
  std::cout << "addTwo(149) = " << callFunct(149, addTwo) << "\n";
  std::cout << "timesFour(4) = " << callFunct(4, timesFour) << "\n";
  return 0;
}
```

7.9. Passing functions to functions: Passing pointer to function

Example04 (difficulty level: 🌶🌶🌶):

```
/** Define the class' methods below so that the main function makes sense */
#include <iostream>      // terminal output
#include <functional>     // use std::function to pass functions as parameter
class NumberSequence {  // class for sequence of whole, positive numbers

 public:
  NumberSequence(uint16_t length = 10);
  // apply the function func() to all numbers:
  void forEach(std::function<uint16_t(uint16_t)> func);
  void print() const;  // print all numbers to console

 private:
  const uint16_t length;  // length of number sequence
  uint16_t *seq;  // the numbers are stored as a dynamic array
};
```

7.9. Passing functions to functions: Passing pointer to function

Example04 (difficulty level: 🌶🌶🌶):

```
// define all NumberSequence methods here
//

uint16_t times2(uint16_t n) { return n*2; }

int main() {
    NumberSequence s;
    s.print();
    s.forEach( &times2 ); // apply the function times2 to all numbers
    s.print();
    return 0;
}
```

7.10. Overview of Memory Segments