Pointers

- We have put off pointers to as late as possible
- Pointers are an essential feature of both C and C++, but are often over-used or misused in C++
- C++ allows us to call by name (References), making much use of pointers unnecessary

⚠️ Pointers give us all the rope we need to hang ourselves, and shoot ourselves in the foot, and blow our leg off.

How pointers work:

- A variable name is a human-friendly shorthand for its address.
- The assignment: `x = 17;` is short for:
  
  Put the value “17” into the memory location labelled “x”.
- If our computer has 256 MB of (virtual) memory, then x’s location could be a byte offset from 0 to $2^{28}$.

Main uses of pointers in C++

1. Arrays
2. Dynamic memory (and object) allocation
3. The current this object
4. Fancy footwork

But first the syntax. There are 2 operators:

1. The “address of” operator, &
   
   If x is any valid variable, then $&x$ is the address of x.
   
   `#include <iostream.h> // addressof.cc`

   ```cpp
   int main() {
     int x=17;
     cout << "x" << dec << x << "", $&x$ << dec << $&x$ << endl;
     return 0;
   }
   ```

2. The “dereference” (pointer) operator, *
   
   If p is any valid address (pointer), then *p is the content of that address.
   
   `#include <iostream.h> // addressofint.cc`

   ```cpp
   int main() {
     int x=17;
     int* px=&x;
     cout << "p" << dec << (int)px << ", px" << dec << *px << endl;
     return 0;
   }
   ```

Read a statement such as int* px; backwards:

"px is a pointer to an int"
A pointer is a distinct type. A pointer of type int* is a different type than a float*

```
#include <iostream.h> // addressoffloat.cc

int main() {
    int x=17;
    int* px=&x;
    float e=1.6e-19;
    float* pe=&e;
    cout << "px= " << hex << (int)*px << ", *px= " << dec << *px << endl;
    cout << "pe= " << hex << (int)*pe << ", *pe= " << dec << *pe << endl;
    return 0;
}
```

```
#include <iostream.h> // const.cc

int main() {
    const int x(17);
    int y(42);
    int* p1;
    const int* p2;
    const int* const p3(&x);
    int* const p4(&y);
    p1 = &y;
    *p1 = 2;
    p2 = &x;
    (*p4)++;
    cout << "p1= " << hex << (int)p1 << ", *p1= " << dec << *p1 << endl
         << "p2= " << hex << (int)p2 << ", *p2= " << dec << *p2 << endl
         << "p3= " << hex << (int)p3 << ", *p3= " << dec << *p3 << endl
         << "p4= " << hex << (int)p4 << ", *p4= " << dec << *p4 << endl;
    return 0;
}
```

**const pointers and pointers to const**

Before we look at the use of pointers, we’ll clear up (or add) another source of confusion:

- If we declare `const int x(17);` then `x` is immutable—it cannot be changed.
- With pointers, we can make either the pointed-to object constant, or the pointer itself constant, or both:
  - `int* p1;` // `p1` and `*p1` mutable
  - `const int* p2;` // `p2` mutable, `*p2` immutable
  - `const int* const p3(&x);` // `p3` and `*p3` immutable
  - `int* const p4(&y);` // `p4` immutable, `*p4` mutable

Any of:

- `int* pi;
- int *pi;
- int*pi;

are syntactically correct.

We prefer: **int* pi;** to emphasize that the type of `pi` is a pointer to an int.
**Pointer Arithmetic**

- We can do pointer arithmetic in a way that makes sense for that pointer.
- if \( p \) is an int*, then \( p+1 \) points to the first byte after \( p \).
- The compiler looks after the different variable lengths: we don’t have that worry.

```cpp
#include <iostream.h>  // arithmetic.cc

int main() {
    int x(17);
    int y(42);
    int* p1(&x);
    cout << "p1 = \&y = " << (int*)(p1+1) << endl;
    cout << "p1 = " << hex << (int)p1 << ", *p1 = " << dec << *p1 << endl;
    ++p1;
    cout << "p1 = " << hex << (int)p1 << ", *p1 = " << dec << *p1 << endl;
    // (*p1)++;  // this will change random memory
    return 0;
}
```

**Notes:**

- It looks like the difference of the 2 pointers is 4, whereas pointer arithmetic says it's 1. This is consistent.
- Pointers allow us to access memory that maybe we shouldn’t access
- Sometimes this leads to the dreaded segmentation fault
- Be careful with the precedence of * and ++. When in doubt, use parentheses.

> **Be very careful with pointer arithmetic**

**Arrays**

- Arrays use pointers implicitly — and interchangeably
- The array name is a pointer to the 0th element of the array
- Then name+1 is a pointer to the 1st element of the array, etc.
- We can use either indexed arrays, or pointers.
```cpp
#include <iostream.h> // array.cc

int main() {
    const int kArraySize(8);
    const int a[]={0, 1, 4, 9, 16, 25, 36, 49};
    const int* p(a);
    for (int i=0; i<kArraySize; i++) {
        cout << *p++ << endl;
    }
    return 0;
}
```

**Points to note:**

- The declaration: `const int a[]` makes an array, of size determined by the initialization,
- The definition `{0, 1, 4, 9, 16, 25, 36, 49}` sets the size of the array, and initializes its elements
- The declaration: `const int* p(a);` declares a pointer to `const int`
- We can access each element with: `*p++` (first dereference, then bump the pointer).

---

**the arguments of main**

- So far, we have used `main()` with no arguments.
- It can also take 2 arguments: `main(int argc, char* argv[])`. (By convention, these names are used, but they can be anything.)
  - `int argc`
    - The number of command line arguments. The program name is always the first argument, so `argc>0`
  - `char* argv[]`
    - A NULL-terminated array of pointers to `char[]`. Each array of `char` is a NULL-terminated string containing the argument.
- (See K & R for a description of arrays, and the use of `argc, argv`)
We can also use the interchangeability of pointers and arrays:

```cpp
#include <iostream.h>  // arguments2.cc

int main(int, char** argv) {
    while (*argv != 0) {
        cout << *argv++ << endl;
    }
    return 0;
}
```

- Pointers allow us to be very terse (and cryptic)
- We can use either `char* argv[]` or `char** argv`
- The `argv` array is terminated with a NULL pointer.

---

**Dynamic object allocation**

- So far, we have declared every object (including built-in types) at compile-time
- Very often, we want to wait until runtime
  - We may want to reduce the executable size
  - More normally, we don’t know how big to make the object at compile time
- C++ allows us to do this with the `new` operator

---

**the new operator**

- The `new` operator is a way of creating an object — or dynamically allocating memory.
- It creates the object, and returns a pointer to the object.
- That’s why we simply cannot avoid pointers
- We can create either a single object, or an array of objects

```cpp
#include <iostream.h>  // newobject.cc
#include <cassert.h>
#include "Point7.hh"

int main() {
    Point* p = new Point(5,7);
    assert (p!=0);
    (*p).print();
    p->removeTo(Point(2,1));
    p->print();
    cout << endl;
    return 0;
}
```
Points to Note:

- We use the assert macro to “assert” some condition:
  - if the argument to assert is “true”, nothing happens
  - otherwise, the program terminates with an error message
- The statement: Point* p = new Point(5,7);
  - creates a new Point object, calling the constructor as normal
  - returns a pointer to the object
- If new fails, p is assigned to 0, otherwise it points to the new object
- You should always test that new was successful
- Since p is a pointer, we have to dereference it before using the object
- The operation: (*p).method is done so often, we use the shorthand: p->method

```
#include <iostream.h>  // newarray.cc
#include <strstream.h>
#include <assert.h>

int main(int argc, char** argv) {
  assert (argc>1);
  istringstream arg(argc+arg);
  int arraySize;
  arg >> arraySize;
  int* a = new int[arraySize];
  assert (a!=0);
  for (int i=0; i<arraySize; i++) { a[i] = i*i; }
  const int* p(a);
  for (int i=0; i<arraySize; i++) { cout << *p++ << endl; }
  return 0;
}
```

the delete operator

There is a catch using new:

- Normally, when an object goes out of scope, its memory is freed
- If an object is created with new, memory is not freed

This leads to the dreaded memory leak

We have to use the delete operator to free the memory explicitly:

- Use delete foo to delete an object
- Use delete [] foo to delete an array

Points to Note:

- We use istream to parse the arguments
- This time, new is used to dynamically create an array
- The array size is given with the [] operator
- Since a is int*, we can use an array index or a pointer interchangeably
- We could have used any object to create e.g. an array of Point objects
#include <iostream.h> // deletearray.cc
#include <stdlib.h>
#include <assert.h>

int main() {
    const int kRandMax(1024);
    while (1) {
        const int kArraySize( (rand()%kRandMax)+1 );
        int* a = new int[kArraySize];
        assert (a!=0);
        for (int i=0; i<kArraySize; a[i++]=rand()) {}
        cout << "a = " << hex << (int)a
             << "", kArraySize = " << dec << kArraySize << endl;
        delete [] a;
    }
    return 0;
}

Points to Note:

- **Without** the `delete`, the array starts at a different address each iteration — memory is consumed until the program crashes
- **With** the `delete`, the memory is recycled (try it)

![Always make sure to use delete after using new]

Object Destructor

- When an object goes out of scope, its memory is freed
- If we used a `new` operator for memory that is part of the object, that memory cannot be freed, unless we do so explicitly
- Every class has a default destructor that is called when the object goes out of scope
- But the default cannot know about memory allocated dynamically — so we have to provide an explicit destructor
- If the class is called `Foo`, the destructor is called *`Foo*"

![Always provide a destructor when memory has been allocated dynamically]
Some subtle points:

- In the first example, the destructor is called even though we haven't called it.
- In the second example, the destructor is not called unless we explicitly call it.
- This last destructor does nothing—we'll now look at one that does something essential.

```cpp
#include <iostream.h> // destructor2.cc
#include <stdlib.h>
#include "PointB.hh"

Point::Point() {
    cout << "I am the destructor" << endl;
}

int main() {
    const int kRandMax(1024);
    for (int i=0; i<20; i++) {
        Point* p = new Point( rand()%kRandMax, rand()%kRandMax );
        cout << "Point " << i << ": ";
        p->print();
        cout << endl;
        delete p;
    }
    return 0;
}
```

---

In this example, dynamic memory is allocated by the constructor:

```cpp
#define __FOOBAR_HH // FooBar1.hh
#define __FOOBAR_HH
#include <iostream.h>

class FooBar {
public:
    FooBar(unsigned int size=0, int initValue=0);
    ~FooBar();
    void print();
private:
    unsigned int m_size;
    int* m_array;
};
#endif // __FOOBAR_HH
```

```cpp
#include "FooBar1.hh" // FooBar1.cc

FooBar::FooBar(unsigned int size, int initValue) : m_size(size) {
    m_array = new int[m_size](initValue);
}

FooBar::~FooBar() {
    delete [] m_array;
    cout << "I have destroyed you" << endl;
}

void FooBar::print() {
    cout << "array size = " << m_size << ": ";
    for (unsigned int i=0; i<m_size; i++) {
        cout << " " << m_array[i];
    }
    cout << endl;
}
```
#include <iostream.h> // destructor3.cc
#include <stdlib.h>
#include "FooBar1.hh"

int main() {
    const int kRandMax(16);
    for (int i=0; i<20; i++) {
        FooBar f( rand()%kRandMax, rand() );
        f.print();
    }
    return 0;
}

Note that:
- Our destructor is called automatically
- The scope of the object is within { }

---

default constructors

As well as a default destructor, C++ also provides 2 default constructors:

1. a “bare” constructor
2. a copy constructor – its signature is: Foo(Foo&) or Foo(const Foo&)

i.e, if we don’t write them, the compiler provides them.
If we don’t allocate memory, there is usually not a problem:
- the “bare” constructor may or may not initialize the data members
- the copy constructor copies all data members (recursively)

---

#include <iostream.h> // default.cc

class Foo {
    public:
        void print() { cout << m_x << endl; }
    private:
        int m_x;
;
    int main() {
        Foo f;
        Foo g(f);
        f.print();
        g.print();
        return 0;
    }

---

but if we allocate memory in the object, we could be in for a surprise

#include <iostream.h> // copyconstructor1.cc
#include <stdlib.h>
#include "FooBar2.hh"

int main() {
    const int kRandMax(8);
    for (int i=0; i<10; i++) {
        FooBar f( rand()%kRandMax, rand() );
        f.print();
        FooBar g(f);
        f.increment();
        g.print();
    }
    return 0;
}
#include "FooBar2.hh" // FooBar2.cc

FooBar::FooBar(unsigned int size, int initValue) : m_size(size) {
    m_array = new int[m_size](initValue);
}
FooBar::FooBar() {
    delete [] m_array;
    cout << "I have destroyed you" << endl;
}
void FooBar::print() {
    cout << "array size = " << m_size << ":";
    for (unsigned int i=0; i<m_size; cout<<"<<m_array[i++]") {} 
    cout << endl;
}
void FooBar::increment() {
    for (unsigned int i=0; i<m_size; m_array[i++]=) {} 
}

#ifndef __FOOBar_HH  // FooBar3.hh
#define __FOOBar_HH
#include <iostream.h>

class FooBar {
public:
    FooBar(unsigned int size=0, int initValue=0);
    FooBar(const FooBar&);  // copy constructor
    "FooBar();
    void print();
    void increment();
private:
    unsigned int m_size;
    int* m_array;
};
#define __FOOBar_HH

#include "FooBar3.hh" // FooBar3.cc
FooBar::FooBar(unsigned int size, int initValue) : m_size(size) {
    m_array = new int[m_size](initValue);
}
FooBar::FooBar(const FooBar& f) {
    cout << "Copy constructor ..." << endl;
    m_size = f.m_size;
    m_array = new int[m_size];
    for (unsigned int i=0; i<m_size; i++) {m_array[i]=f.m_array[i];}
}
FooBar::"FooBar() {
    delete [] m_array;
    cout << "I have destroyed you" << endl; }
void FooBar::increment() {
    for (unsigned int i=0; i<m_size; m_array[i++]=) {} }
void FooBar::print() {
    cout << "array size = " << m_size << ":";
    for (unsigned int i=0; i<m_size; cout<<"<<m_array[i++]) {} 
    cout << endl; }

Don't count on the default constructors - provide explicit ones

The problem is:

- The default copy constructor does a member-by-member copy
- The pointer m_array is copied to the new object, so in g it points to the same data
- Instead, we have to allocate new memory for the new object
#include <iostream.h>  // copyconstructor1.cc
#include <stdlib.h>
#include "FooBar2.hh"

int main() {
    const int kRandMax(8);
    for (int i=0; i<10; i++) {
        FooBar f( rand() % kRandMax, rand() );
        f.print();
        FooBar g(f);
        f.increment();
        g.print();
    }
    return 0;
}
#ifndef __PERSISTENT_HH
   // Persistent.hh
#define __PERSISTENT_HH
#include <fstream.h>

class Persistent {
public:
   void write(ofstream & os) { os.write((char*)this, size()); }
   void read(ifstream & is) { is.read((char*)this, size()); }
protected:
   Persistent() {}
   virtual int size() = 0;
};
#else // __PERSISTENT_HH


#endif // __PERSISTENT_HH

#include "Persistent.hh" // Persistent.cc

void Persistent::write(ofstream & os) {
   os.write((char*)this, size());
}

void Persistent::read(ifstream & is) {
   is.read((char*)this, size());
}


#include "stdlib.h" // PersistentWrite.cc
#include "PooBar4.hh"

int main() {
   PooBar f( rand(), rand(), rand(), rand() );
   f.print();
   ofstream out("myobject.dat");
   f.write(out);
   return 0;
}
and here's the persistent read program:

```
#include "FooBar4.hh" // PersistentRead.cc

int main() {
  FooBar f;
  ifstream in("myobject.dat");
  f.read(in);
  f.print();
  return 0;
}
```

and we can use `new` in the `read` program:

```
#include "FooBar4.hh" // PersistentRead2.cc
#include "Foo.hh"

int main() {
  FooBar f;
  Foo* bar = new Foo;
  ifstream in("myobject.dat");
  f.read(in);
  bar->read(in);
  f.print();
  bar->print();
  delete bar;
  return 0;
}
```

We can see that the virtual function does "the right thing" by adding another persistent class:

```
#include <stdlib.h>  // PersistentWrite2.cc
#include "FooBar4.hh"
#include "Foo.hh"

int main() {
  FooBar f( rand(), rand(), rand(), rand() );
  f.print();
  Foo bar1();
  bar.print();
  ofstream out("myobject.dat");
  f.write(out);
  bar.write(out);
  return 0;
}
```

Exercise for the student:

Repeat this for an object that allocates memory in the constructor

⚠️ There is no shortage of rope with which you can hang yourself