COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Pointers

- A hard concept
- A source of many bugs
- Best avoided where possible
- But sometimes necessary for efficiency

Pointers

- Represents the address of an object
- Every object has a distinct memory address
- C++ allows you to access that address
- Pointers store these addresses
- Can think of an address as an integer

Declaring Pointers

- Pointers are declared with a `*` after the name of the type
- `int *p;`
  - p is a pointer to an int
- `string* p;`
  - p is a pointer to a string

Declaring Pointers

- `double* p, p2;`
- `double *p3, *p4;`
  - p is a pointer to a double
  - p2 is a double (not a pointer!)
  - p3 and p4 are pointers to doubles
  - Don't do this!

Getting an Address

- `&` is the address operator
- It returns a pointer to the object
- In other words the object's address
- Read it in English as "the address of"
- Eg:
  - `int i = 42;`
  - `int *p = &i;`
- "Place the address of i into p"
- "Make p point to i"

Pointer Example

```
int main() {
   int i = 1;
   double d = 2.3;
   std::string s = "4";
   std::cout << "&i : " << &i << endl;
   std::cout << "&d : " << &d << endl;
   std::cout << "&s : " << &s << endl;
}
```

Dereferencing

- * is the dereference operator
- It gets the object a pointer is pointing to
- Opposite to the & operator
- Eg:
  - int i = 42;
  - int *p = &i;
  - int j = *p; // equiv to: int j = i;

Dereferencing Example

```
int main() {
   int i = 42;
   int *p = &i;
   std::cout << i << ' ' << *p << endl;
   (*p)++;
   std::cout << i << ' ' << *p << endl;
}
```

- Outputs:

```
42 42
43 43
```

Null pointers

- No variable ever has an address of 0
- Use this to indicate not pointing to anything
- Called "null" pointers
- NULL defined in #include <iostream>
- Dereferencing a null pointer causes a crash
- Can convert pointers to bool
  - NULL is false (since it's 0)
  - Anything else is true

Null pointer example

```
void output(int* p) {
      std::cout << *p << std::endl;
}
int main() {
      int i = 42;
      int *p = &i;
      int *q = NULL;
      output(p);
      output(q);   // CRASH
}
```

Better null pointer example

```
void output(int* p) {
      if (p)
            std::cout << *p << std::endl;
}
int main() {
      int i = 42;
      int *p = &i;
      int *q = NULL;
      output(p);
      output(q);
}
```

Function Pointers

- We can also have pointers to functions
- string (*fp)(int);
  - fp is a pointer to a function
  - The function takes a single int parameter
  - And returns a string
- int (*fp)(void);
  - fp is a pointer to a function
  - It takes no arguments
  - And returns an int

Function Pointer Example

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

```
int main() {
  int x = 1, y = 3;
  void (*funcpoint)(int&, int&);
  funcpoint = &swap;
  swap(a, b);
  std::cout << a << ' ' << b << endl;
  (*funcpoint)(a, b);
  std::cout << a << ' ' << b << endl;
}
```

- Outputs:

```
3 1
1 3
```

Functions

- Can only do two things with a function
  - Call it
  - Take its address with & operator
- Thus two shortcuts are possible
- fp = function; instead of fp = &function;
- i = fp(); instead of i = (*fp)();

Function Pointer Example

```
int main() {
  int x = 1, y = 3;
  void (*funcpoint)(int&, int&);
  funcpoint = swap;
  swap(a, b);
  std::cout << a << ' ' << b << endl;
  funcpoint(a, b);
  std::cout << a << ' ' << b << endl;
}
```

Why Function Pointers

- Allows functions to be:
  - passed as parameters to functions
  - stored in vectors, etc
- Reduces code repetition

Simple example: find_first

```
int find_first(const vector<int> &v,
               bool pred(int)) {
   for (int i = 0; i < v.size(); i++) {
      if (pred(v[i])) {
         return i;
      }
   }
   return -1;
}
```

```
bool even(int i) {
   return (i % 2 == 0);
}
bool positive(int i) {
   return (i > 0);
}

vector<int> v;
std::cout << find_first(v, even) << endl;
std::cout << find_first(v, positive) << endl;
```

Arrays

- A built in container type
- Similar to Java arrays
- Fixed size - can not grow or shrink
- Use `size_t` for indexes
- Use `ptrdiff_t` for distance between elements
- Does not keep track of size
  - So you must (in a size_t variable)
- Declaration example:
  `int an_array[4];`

Array Initialisation

- `int an_array[ ] = {1, 2, 3};`
  - Declares an array of size 3 called an_array
  - Initialises it to contain 1, 2, 3
- `int an_array[5] = {1, 2, 3};`
  - Declares an array of size 5
  - Initialises it to contain 1, 2, 3, 0, 0

Array Initialisation

- `int an_array[5] = { };`
  - Declares an array of size 5
  - Initialises it to contain all zeros
- `int an_array[5];`
  - Not guaranteed to be initialised
  - ie. values could be anything
- `int an_array[2] = {1, 2, 3};`
  - Compile error (too many initialisers)

Array Example

```
const size_t P_Dim = 3;
double point[P_Dim];
*point = 42.0;
```

- Using an array name as a value
  - Gives a pointer to the first element
  - What about the other elements?

Pointer Arithmetic

- We use arithmetic on pointers
- (point + 1) pointer to the 2nd element
- (point + 2) pointer to the 3rd element
- (point + 3) pointer past the end
- We can also use ++, -- and -

Pointer Arithmetic Example

```
const size_t P_Dim = 3;
double array[P_Dim];
double* point;
*array = 42.0;
*(array + 1) = 42.5;
*(array + 2) = 43.0;
point = array + 1;
cout << *(point + 1) << endl;
```

- Outputs 43.0

Indexing

- Pointers support indexing
- point[n] is equivalant to *(point + n)
  - point is a pointer
  - n is an integer
- Since array names act as pointers
  - arrays can be indexed as well
- Typically indexing is done on arrays and not pointers

Indexing Example

```
const size_t P_Dim = 3;
double array[P_Dim];
double* point;
point[0] = 42.0;
point[1] = 42.5;
point[2] = 43.0;
point = array + 1;
cout << point[1] << endl;
```

- Outputs 43.0

Array-based find_first

```
int find_first(const int *v, int n,
               bool pred(int)) {
  for (int i = 0; i < n; i++) {
    if (pred(v[i])) {
      return i;
    }
  }
  return -1;
}
```

```
int v[ ] = {-1, -3, -5, 3, -2};
cout << find_first(v, 5, even) << endl;
cout << find_first(v, 5, positive) << endl;
cout << find_first(v, 3, even) << endl;
cout << find_first(v, 3, positive) << endl;
```

Iterator-based find_first

```
int* find_first(const int* begin,
                const int* end,
                bool pred(int)) {
  while (begin != end && !pred(*begin))
      begin++;
  return begin;
}
```

```
int v[ ] = {-1, -3, -5, 3, -2};
int* p = find_first(v, v + 5, even);
if (p == v + 5) {
    cout << "No even nums found";
} else {
    cout << "First even num: " << *p;
}
cout << endl;
```

String literals

- String literals are actually arrays of chars (ie. `char [ ]` or `char*`)
- This is a hang over from C
- The end of the string is marked by a nul ('\0') char
- This means strings can't contain '\0's

C-Style Strings

- Many library functions deal with them
- `#include <cstring>` contains them
- `size_t strlen(const char* str)`
  - The length of the string
  - Doesn't count the '\0'
- `char* strcpy(char* dst, const char* src)`
  - Copies one string to another
  - Doesn't check if it will fit
- Plus a lot more...

Arguments to main()

- main() actually takes arguments
- `int main(int argc, char** argv)`
- `argc` is the number of elements in `argv`
  - Needed since arrays don't keep track of length
- `argv` an array of char*'s (strings)
- `argv[0]` is the program name
- `argv[1]` to `argv[argc-1]` are command line arguments

main() Example

```
int main(int argc, char** argv) {
  std::cout << "Program name: "
            << argv[0] << std::endl;
  for (int i = 0; i < argc; ++i)
      std::cout << "Arg " << i <<
          " is: " << argv[i] << std::endl;
}
```

Arguments Example

```
bash$ g++ -Wall -g -o argex argex.cc
bash$ argex These are the arguments
Program name: argex
Arg 0 is: argex
Arg 1 is: These
Arg 2 is: are
Arg 3 is: the
Arg 4 is: arguments
```

Memory Allocation

- Memory for variables can be allocated in 3 ways
  - Automatically (local)
  - Statically (globals)
  - Dynamically

Automatic Variables

- Local variables are declared in functions
- Stored on the stack
- Memory is automatically released
- Object disappears when function ends
- Don't keep pointers which are invalid

Automatic Example

```
void do_something(int *p) {
}

int* func() {
   int i = 42;
   do_something(&i);  // OK
   return &i;  // NOT OK!
}
```

Static Allocation

- Declared outside functions (globals)
- Or declared as static within a function
- Not destroyed until the program ends
- Variable is the same every function call

Static Example

```
int* func() {
   static int i = 42;
   i++;
   return &i;
}
int main() {
   int* p1 = func();
   int* p2 = func();
   cout << p1 << " : " << *p1 << endl;
   cout << p2 << " : " << *p2 << endl;
}
```

Dynamic Allocation

- Sometimes we want to create a new object
  - Doesn't disappear at function end
  - Not shared like a static variable
- This is done with dynamic allocation
- Stored on the heap
- Using new and delete

New

- Returns a pointer to a newly created object
- Returns NULL if unable to allocate memory
- new T;
  - New object of type T
- new T(args);
  - Also sets a specific value or passes parameters to constructor

Delete

- Destroys an object allocated by new
- delete ptr;
  - Deletes the object pointed to by ptr
  - Object must have been allocated by new
  - ptr can be NULL (nothing happens)

Dynamic Example

```
int* func() {
  return new int(42);
}
int main() {
  int* p1 = func();
  int* p2 = func();
  cout << p1 << " : " << *p1 << endl;
  cout << p2 << " : " << *p2 << endl;
  delete p1;
  delete p2;
}
```

Dynamic Allocation of Arrays

- new T[n];
  - allocates an array of n T objects
  - returns a pointer to the first element
- delete[ ] ptr;
  - deallocates the array
  - must point to first element of array
  - must have been allocated with new[ ]
  - ptr can be NULL (nothing happens)

Dynamic Array Example

```
int* read_data(size_t n) {
  int* a = new int[n];
  for (size_t i = 0; i < n; ++i)
      std::cin >> a[i];
  return a;
}
```

Dynamic Array Example

```
int main() {
    int* data = read_data(5);
    for (size_t i = 0; i < 5; ++i)
        std::cout << data[i] << std::endl;
    delete[ ] data;
}
```

What about this?

```
int main() {
    int* data;
    data = read_data(5);
    for (size_t i = 0; i < 5; ++i)
        std::cout << data[i] << std::endl;
    data = read_data(5);
    for (size_t i = 0; i < 5; ++i)
        std::cout << data[i] << std::endl;
    delete[ ] data;
}
```

Memory leaks

- The first array created (of size 5)
  - Hasn't been deleted
  - But no pointer points to it anymore
  - Can't be accessed
  - Can't be deleted
  - Is said to have been "leaked"
- Memory leaks are bad
  - Waste memory
  - Potentially use up all memory

Leak-free code

```
int main() {
    int* data;
    data = read_data(5);
    for (size_t i = 0; i < 5; ++i)
        std::cout << data[i] << std::endl;
    delete[ ] data;
    data = read_data(5);
    for (size_t i = 0; i < 5; ++i)
        std::cout << data[i] << std::endl;
    delete[ ] data;
}
```

Avoiding memory leaks

- Make sure that each time memory is allocated with new
  - Somewhere later in the prorgam it is deallocated with delete
- Not always as easy as it sounds