Pointers & Dynamic Memory

Chapter 10

Variables: Address vs Content

Address is intrinsic; content may change

- A left value (aka l-value)
  - is a value that can be on the left hand side of =
  - is the address of a variable

- A right value (aka r-value)
  - is a value that can be on the right hand side of =
  - can be any valid expression (i.e., constant, variable, function call, or any of these joined with operators and grouped with parentheses)

  i = 10;  // i is an l-value, 10 is an r-value
  j = i;   // j is an l-value, i is an r-value

  i  10  0xfffff00
  r-value of i is 10 (i.e., the content of i)
  l-value of i is 0xfffff00 (i.e., the address of i)

Pointer and Address Operators

Pointers are variables that contain memory addresses

- * pointer-type specifier (variable definitions & declarations)
  - type t;
  - type p;    // same as above
  - type *p;   // same as above

- & address of operator returns the address of variables
- * dereference or indirect operator (expressions)

- Operator relationships
  - *=p is assignment compatible with t (i.e., t = *p)
  - &= is assignment compatible with p (i.e., p = &i)
  - *(variable) = variable

- Special pointers (that point to nothing)
  - NULL
  - 0 (zero)

Address Operators

Memory content vs memory address

  int i;
  i  undefined  0x0a000010 i is undefined

  int* p = &i;
  p  0x0a000010  0x0a000014 p is 0x0a000010

  i = 10;
  i 10  0x0a000010 i is 10

  *p is now 10
  p  0x0a000010  0x0a000014 p is 0x0a000010

  *p is 10

Pointer Syntax

Simple examples

  int i;
  int* ip1;
  int* ip2;

  i = 10;
  ip1 = &i;    // *ip1 == 10
  ip2 = ip1;   // *ip2 == 10

  cout << *ip1 << *ip2 << endl;    // prints addresses
  cout << *ip1 << *ip2 << endl;    // prints variable values

  ip1 = 0;    // points at nothing
  ip2 = NULL; // ditto

  while (ip1 == NULL) ....
  while (ip2 != 0) ....
  if (ip1 == ip2) ....
  if (ip1 != ip2) ....

Pointer Arithmetic

Addition, subtraction, and comparison

- Assumes a contiguous block of memory (i.e., an array)
- Allowed operations
  - Add/Subtract an integer to/from a pointer: p1 ± i = p2
  - Difference between two pointers: p2 - p1 = i (# of elements)
  - Test for (inequality): if (p1 == p2) / if (p1 != p2)

- No check for addressing out of bounds

- The compiler automatically multiplies and divides by the size (in bytes) of the referenced object in pointer arithmetic

  p5
  p2
  p1

  p5 - p1 = 4
  p2
  p1
  p5
**Arrays As Function Arguments**

Arrays are always passed by address

- Arrays are the exception: always passed by address
- Function definitions
  - void func1(int student[]) {...} // most common
  - void func1(int student[10]) {...} // least common
  - void func2(float test[] [4]) {...}
  - void func3(double class[]) [4] [5] {...}

- Function calls
  - func1(test);
  - func2(test_score);
  - func3(class_score);

**Pass By Address Illustrated**

Efficient function calls

```c
int test[10];

void func1(int* student)
{
    // student[6] = 95;
    // 76 95
    // ... 83
    // 100
    // ... 89
    // ... 91
    // ... 89
    // ... 97
    // ... 41
    // ... 0
}

int main(int argc, char* argv[]) {  // arg demo hello world from CS1130
    for (i = 1; i < argc; i++)
        printf("%s", argv[i]);

    // argv[0] = arg demo
    // argv[1] = hello
    // argv[2] = world
    // argv[3] = from
    // argv[4] = CS1130
    // argv[5] = NULL
}
```

**Returning Arrays**

Returning pointers (a common interview test)

// technique 1 – WRONG
```c
char* getData()
{
    char* line[80];
    ... return line;
}
```

// technique 3 – okay
```c
char* getData(char* line)
{
    ... return line;
}
```

// technique 2 – okay
```c
char* getData()
{
    char* line = new char[80];
    ... return line;
}
```

// technique 4 – okay
```c
char* getData()
{
    static char* line[80];
    ... return line;
}
```

**Command-Line Arguments**

Arrays of pointers

```c
int main(int argc, char* argv[])
{
    for (i = 1; argc[i] != NULL; i++)
        printf("%s", argv[i]);

    for (i = 1; i < argc; i++)
        printf("%s", argv[i]);
}
```

**Pointers To Functions**

Dynamic function manipulation

- The address of a function is its entry point
- The name of the function, without any adornment, is its address
- "Configurable" algorithms

```c
fp("Hello World");
fa[1]("Hello World");
```

**User-Specified Exit Handlers**

ANSI atexit function (an example of a function pointer)

```c
#define _EXITARGS 2
#define _EXIT_VALUE 0
int atexit(void (*func)(void));
```

- prototype located in <stdlib.h>
- func is a pointer to a function returning void and taking no parameters
- The name of a function, by itself, is a pointer
- User can register up to 32 exit handlers
  - functions are called in reverse order of registration
  - functions are called as many times as they are registered
**atexit Demo**

Functions as parameters

```c
#include <stdlib.h>
#include <stdio.h>

void cleanUp(void) {
    printf("Function cleanUp called\n");
}

void shutDown(void) {
    printf("Function shutDown called\n");
}

void main(void) {
    atexit(shutDown);
    atexit(cleanUp);
    exit(0);
}
```

**Pointers To Pointers**

Multiple indirection

- Useful for allowing a function to change an argument that is already a pointer
- Used to write qsort comparison functions

```c
int i1, i2;
int *p;
int **pp;
// or int *p[ ]

void func(int **pp) {
    cout << **p;
    *pp = &i2;
    cout << *p;
}

void main(void) {
    int *p;
    p = &i1;
    cout << *p;
    func(&p);
    cout << *p;
}
```

**The ANSI qsort Library Function**

An implementation of the quick sort algorithm

```c
void qsort(void* base, size_t num, size_t size, int (*icmp)(const void*, const void*))
```

- base: address of first element in an array
- num: number of elements in the array
- size: the size in bytes of each array element
- icmp: a pointer to a function, which compares two array elements passed in as two void pointers and returns an integer
  - less than 0 (elements are already in ascending order)
  - 0 (elements order the same)
  - greater than 0 (elements are out of order)
  - modeled after the strcmp function
- #include <stdlib.h>

**Dynamic Memory Allocation**

Run-time memory usage

- Used when the size or number of objects is unknown
- Allocated as a contiguous block from the heap
- The address of the beginning of the block is returned
- NULL (or 0) is returned if the memory was not available
- Remains allocated until explicitly deallocated
- Becomes "garbage" if the address is lost (a memory leak)
- C++ manages dynamic memory with new and delete
  - Creation and destruction of single objects and arrays of objects
  - Creation and destruction of blocks (arrays) of primitive data types

**Memory Layout**

Data storage by class

- high memory: dynamic variables
- low memory: text
- stack: uninitialized data, initialized data
- machine instructions: parameters/automatic variables

**Dynamic Object Creation**

Dynamic instantiation (vs static instantiation)

- `new`
  - Allocates memory on the heap
  - Calls the constructor for the new object
  - Returns the address of the object and does the equivalent of a type cast (i.e., the pointer is cast to a pointer of the appropriate type)
- `delete`
  - Calls the destructor
  - Dealocates the memory pointed to by the operand and returns it to the available pool on the heap
new vs malloc

- Dynamic memory C style
  - `int* ip = (int*) malloc(100 * sizeof(int));`
  - `struct A* ptr = (struct A*) malloc(sizeof(struct A));`
  - `free(ip);`
  - `free(ptr);`

- Dynamic memory C++ style
  - `int* ip = new int[100];`
  - `A* ptr = new A;` // same syntax for a class
  - `delete ip;`
  - `delete ptr;`

new and delete Example

Instantiating and destroying objects

class foo
{
  public:
    foo(); // #1
    foo(int); // #2
    int func();
};

foo* fp1 = new foo(); // #1
foo* fp2 = new foo(100); // #2
foo* fp3 = new foo(100); // #1 * 100
delete fp1;
delete fp2;
delete []fp3;

Member Access

Which constructor is called?

```cpp
foo f;
// static instantiation
f.func();

foo* fptr = new foo();
// dynamic instantiation
fptr->func();

foo* fa1 = new foo[100];
fa1[50].func();

foo* fa2[100];
for (int i = 0; i < 100; i++)
  fa2[i] = new foo[i];
for (int i = 0; i < 100; i++)
  cout << fa2[i]->func() << endl;
```

Copy and Assignment (Simple Class)

Classes that do not contain pointers

```cpp
struct point
{
  int x;
  int y;
};

class triangle
{
  private:
    point* v1; // composition
    point* v2;
    point* v3;
  public:
    triangle& operator=(triangle& t);
    triangle& triangle(t); // compiler
};
```

Copy and Assignment (Complex Class)

Classes that contain pointers

```cpp
struct point
{
  int x;
  int y;
};

class triangle
{
  private:
    point* v1; // aggregation
    point* v2;
    point* v3;
  public:
    triangle& operator=(triangle& t);
    triangle& triangle(t); // copy ctor – supplied by the compiler
};
```
Copy and Assignment (Version 2)

Desired results

```cpp
struct point {
    int x;
    int y;
};

class triangle {
    private:
        point* v1; // aggregation
        point* v2;
        point* v3;
    public:
        triangle(triangle& t); // user
        triangle& operator=(triangle& t);
};

class overridden copy or assignment (copies objects)
```

```cpp
v1
v2
v3
```

Copy Constructor and operator=

Overriding the defaults

```cpp
// copy constructor
triangle::triangle(triangle& t) {
    v1 = new point;
    v2 = new point;
    v3 = new point;
    if (v1 == 0) delete v1;
    if (v2 == 0) delete v2;
    if (v3 == 0) delete v3;
    v1->x = t.v1->x;
    v1->y = t.v1->y;
    v2->x = t.v2->x;
    v2->y = t.v2->y;
    v3->x = t.v3->x;
    v3->y = t.v3->y;
}
```

```cpp
v1
v2
v3
```

```cpp
// overloaded assignment operator
triangle& triangle::operator=(triangle& t) {
    if (&t == this) {
        return *this;
    }
    v1 = new point;
    if (v1 == 0) delete v1;
    if (v2 == 0) delete v2;
    if (v3 == 0) delete v3;
    v1->x = t.v1->x;
    v1->y = t.v1->y;
    v2->x = t.v2->x;
    v2->y = t.v2->y;
    v3->x = t.v3->x;
    v3->y = t.v3->y;
    return *this;
}
```

Copy Constructor & operator= Tasks

A checklist for programmers

- **Copy Constructor**
  - Argument must be pass by reference
  - Member by member copy

- **operator= (Assignment Operator)**
  - Argument is typically pass by reference
  - Test for assigning an object to itself (T = T)
  - Test for previous pointer values and delete if necessary (if the object being assigned to had a previous value, don’t let it become garbage)
  - Member by member copy
  - Return a reference to the new object for chaining (T3 = T2 = T1)

Linked Data Structures

Organizing data with dynamic memory and pointers

- **circularly linked list**
- **binary tree**
- **linked list**