Pointers and Dynamic Memory
(Chapter 7)

Computer Memory

- **Bit** (binary digit): a single boolean value
- **Byte**: eight bits
- Memory is a big array of bytes:

```
char memory[ 67108864 ];
```

- Each byte has two properties
  - A value
  - A location (address)
- A programming language is a high-level abstraction for manipulating (part of) this array

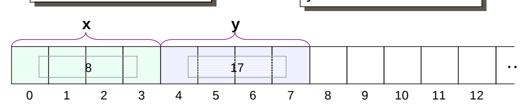
Storing Data in Memory

- Program data can be placed in one of three kinds of memory
  - **Static** memory: space set aside by the compiler ahead of time (don't confuse with C/C++ `static` keyword)
  - **Automatic** memory: space used by local variables of functions during execution
  - **Dynamic** memory: extra space requested explicitly at runtime

Static Memory

```
#include <iostream.h>

int x;
int y;

int main( void )
{
    x = 8;
    y = 17;
    cout << x << " "
         << y << endl;
    return 0;
}
```

```
#include <iostream.h>

char memory[ 67108864 ];
#define x memory[0□3]
#define y memory[4□7]

int main( void )
{
    x = 8;
    y = 17;
    cout << x << " "
         << y << endl;
    return 0;
}
```



Disadvantages of Static Memory

- So far, all programs have a fixed upper bound on size
- This is bad!
  - User input usually not known in advance
  - Different uses of the same program may require different amounts of memory
  - Operating systems are smart enough to give you more memory if you need it
- Need a way to request more memory if needed

Pointers

- If we're going to request more memory, we need a way to refer to it
  - Location of new memory not known in advance
- # **A pointer is an abstraction of an address in memory**

Pointers in C++

- For any type `T`, `T*` is the type "pointer-to-`T`"

```
int *x;                  // A pointer to in integer
Student *pCraig;         // A pointer to a Student
Screen *dynamicScreen;   // A pointer to a Screen
```

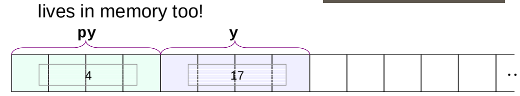
- The placement of the `*` is irrelevant
  - But the book gets it wrong

```
int* x, y; // Not what you expect!
int *x, y; // A bit more clear

int *x;    // This is probably
int y;     // the best
```


The Secret Lives of Pointers

- Recall
  - Memory is just a big array of bytes
  - A pointer is an abstraction of a memory location
- So: a pointer is an index into the memory array
- A pointer is itself a variable, so it lives in memory too!

```
int *py;
int y;

int main( void )
{
    y = 17;
    make py point to y;
    return 0;
}
```



Dereferencing Pointers

- Presumably, a pointer-to-`int` points to an `int`
  - Need a way to get back the actual `int`
- `*` is the dereference operator (not multiplication!)
  - If `p` is of type pointer-to-`T`, then `*p` is of type `T`
  - `(*p)` acts just like a declared variable of type `T`
  - `(*p)` is like `memory[p]`

```
int main( void )
{
    int y = 17;
    int *py;

    make py point to y;

    (*py) = 29;
    (*py) += 13;

    cout << y << " " << (*py)
         << endl;

    return 0;
}
```


Pointers to `struct`s and `class`es

- If `p` points to a struct or class, `.` syntax can be used on `(*p)` to get at member data and functions
- But `->` syntax is a convenient shorthand
  - `ptr->member` is short for `(*ptr).member`

```
#include "student.h"

int main( void )
{
    Student *ps;
    …
    cout << ps->getGPA() << endl;
    return 0;
}
```


The Address of a Variable

- All variables live in some segment of memory
  - So it should be possible to get pointers to them
- Use the `&` operator to get the address of an object
- `&x` returns an index into the memory array that can be used to point to `x`
- Can you take the address of a pointer?

```
int main( void )
{
    int y = 17;
    int *py = &y;

    (*py) = 29;
    (*py) += 13;

    cout << y << " " << (*py)
         << endl;

    return 0;
}
```


Pointers and Arrays

- Arrays are **already** pointers
  - An array variable contains the address of the start of the array
  - Array indexing is equivalent to "pointer arithmetic"
  - All pointers can transparently point to a single object or an array of objects

```
int main( void )
{
    int y = 17;
    int w[] = { 1, 4, 5, 8 };

    int *p;

    p = &y;       // p points to y
    p = w;        // p points to w[0]
    p = &(w[2]);  // p points to w[2]
    p = w + 3;    // p points to w[2]

    return 0;
}
```

Aside: **`scanf`**

- Remember how you needed the mysterious **`&`** in **`scanf`**?

```
#include <stdio.h>
…
int x;
scanf( "%d", &x );
…
```

- C has no reference parameters, so **`scanf`** works using pointers:

```
void scanf( char fmt[], int *x )
{
    int y = read an int from the user;
    (*x) = y;
}
```


Aside: References

- References are just "pretty" pointers

```
void swap( int& x, int& y )
{
    int tmp = x;
    x = y;
    y = tmp;
}

…
int a = 15, b = 19;
swap( a, b );
…
```

⇒

```
void swap( int* x, int* y )
{
    int tmp = (*x);
    (*x) = (*y);
    (*y) = tmp;
}

…
int a = 15, b = 19;
swap( &a, &b );
…
```

- Of course, you should still use references when appropriate

Getting More Memory

- For any type **`T`**, the expression **`new T`** returns a pointer to a *fresh* instance of **`T`**
  - At run time, an unused block of memory is chosen and initialized
  - The address of the block is returned
- Of course, might need to call a constructor of **`T`**
  - Can call constructors in similar way to variable declarations
  - If no arguments are supplied, default constructor is called

Using **`operator new`**

```
#include "fraction.h"
#include "screen.h"

int main()
{
    int *x = new int;
    int *y = new int( 17 );

    Screen *scr1 = new Screen();
    Screen *scr2 = new Screen( 't' );
    scr1->horizontalLine( 3, 6, 12, '*' );

    Fraction *frac = new Fraction( 12, 19 );
    …
    return 0;
}
```


Creating New Arrays

- **`operator new`** can also be used to create arrays
- Arrays can have unspecified size at compile time!
- Use the **`new T[ size ]`** syntax
- Must call default constructor

```
int main( void )
{
    int num;
    int *data;

    cout << "How many items?" << endl;
    cin >> num;
    data = new int[ num ];
    for( int idx = 0; idx < num; ++idx ) {
        cin >> data[ idx ];
    }
    sort( data, num );
    for( int idx = 0; idx < num; ++idx ) {
        cout << data[ idx ];
    }
    return 0;
}
```


Dynamic Arrays

- Dynamically allocated arrays are a powerful tool

```
#ifndef __SCREEN_H__
#define __SCREEN_H__

class Screen {
public:
    Screen( int width, int height );

    void putChar( int col, int row, char ch );
    char getChar( int col, int row );
    …
private:
    char *data;
    int width;
    int height;
};

#endif
```

The Heap

- Dynamically allocated memory comes from "the heap"
  - A chunk of memory set aside just for dynamically created objects

Cleaning Up

- Just as dynamic memory must be explicitly allocated, it must be explicitly deallocated

```
void blarg()
{
    int *a = new int;
}

int main( void )
{
    for( int idx = 0; idx < 1000000; ++idx ) {
        blarg();
    }
    return 0;
}
```

- Need a way to tell dynamic memory to go away

The **`delete`** operator

- The statement **`delete p`** returns the memory associated with **`(*p)`** to the heap
  - No longer legal to refer to **`(*p)`**!
  - Memory can now be reallocated and used for other purposes
  - Works the same way for arrays
- Making sure that **`delete`**s match with **`new`**s is very hard
  - A very common source of bugs
  - Some languages don't have **`delete`**

Using **`delete`**

```
int main( void )
{
    int num;
    int *data;

    cout << "How many items?" << endl;
    cin >> num;
    data = new int[ num ];
    for( int idx = 0; idx < num; ++idx ) {
        cin >> data[ idx ];
    }
    sort( data, num );
    for( int idx = 0; idx < num; ++idx ) {
        cout << data[ idx ];
    }

    delete data;
    return 0;
}
```


Hazards of Dynamic Memory

- Pointers are distinct from what they point to
  - Problems occur when they're not kept in synch
- **Memory leak**: allocated object, but no way to reference it
- **Dangling pointer**: pointer still points to memory location, object no longer exists

```
void memoryLeak()
{
    int *pi = new int;
    pi = new int;
}
```

```
void danglingPointer()
{
    int *pi = new int;
    delete pi;
    (*pi) = 15;
}
```


Summary

- Dynamic memory is memory that is explicitly requested at run time
- A pointer is an abstraction for a memory location
- Dereference and address-of operators
- Pointers and arrays
- Creating new objects and arrays using **`new T`**
- Deleting heap-allocated objects using **`delete ptr`**
- Memory leaks, dangling pointers