Review of Data Types

Integer types: int, long long, char, bool, enum:

- unsigned: stored as binary
- signed: two's complement

Floating-point types: float, double

- sign bit, exponent, mantissa
- double has more bits in mantissa (higher precision)
- round-off errors: don't use == or !=

Arrays: collection of objects of the same type

Classes/structs: collection of objects of possibly different types

Memory Addresses

- Each variable is stored in memory
- The memory consists of a number of cells, each with a unique number called the **address**
- The number of bytes used depends on the type of variable (e.g. char = 1 byte, int = 4 bytes on our machines)
- To find out how many bytes used by a variable, used the `sizeof` operator:

  e.g. `sizeof(int)`, `sizeof(double)`
- Since there may be multiple programs running at the same time, the exact memory address of a variable may be different each time a program is run. We cannot use absolute addresses inside our programs.

Types of Main Memory

Stack:

- grows and shrinks at the "top"
- local variables are "pushed" onto the stack when the function starts (allocation)
- local variables are "popped" off the stack when the function exits (deallocation)
- handled automatically by compiler
- number of local variables is fixed at compile time

Heap:

- memory allocated and deallocated dynamically
- programmer specifies when (and how much) to allocate and deallocate
- memory allocated until explicitly deallocated
- allows size to be determined at run time

The & Operator

- Take the address of a variable:

```
int i;
cout << &i << endl;   // prints the address of i
```

- What if we want to store the address in a variable?
- We need **pointers**

Pointers

- A **pointer** is a variable that stores the address of a variable.
- To declare a pointer, we also need to specify which data type it points to (so it knows the size):

```
int *iPtr;          // pointer to an integer
char *c1, *c2;      // pointers to characters

iPtr = &i;
c1 = &c;
c2 = c1;
```

- Special address `nullptr` (old C++: 0 or `NULL`): means pointer to nothing
- Note: pointers (like other variables) are uninitialized when declared and can point to anywhere

Dereferencing Pointers

- We can refer to the variable pointed to by a pointer using the dereferencing operator `*`:

```
int a = 10;
int *p1, *p2;

p1 = p2 = &a;
*p1 = 15;                  // lvalue: can be assigned to
cout << *p2 << endl;       // aliasing: changing *p1 changes
                           // *p2 too!
```

- To dereference a pointer to a class and refer to its members, use `(*p).f()` or `p->f()` as a shorthand.

Constant Pointers, Pointers to Constants

```
const int a = 100, b = 200;     // constants
int c = 300;
const int *pa = &a;             // pointer to constant
int *const pc = &c;             // constant pointer

*pa = 10;                       // Error
pc = &b;                        // Error
pa = &b;                        // Okay
*pc = 10;                       // Okay
```

Arrays and Pointers

- Arrays are treated the same way as constant pointers—name of array points to the first element
- If we have `int A[10];` Then:

  `*A` is the same as `A[0]`

  `*(A+i)` is the same as `A[i]` (* has higher precedence than +)
- `A` is the same as `&A[0]`

  `A+i` is the same as `&A[i]`
- When passing arrays as parameters, `int A[]` is the same as `int *const A`
- Can also do `p++`, `p--`, `p-q`. Be careful with the last one!
- Pointer arithmetic can be "dangerous"

Allocating Memory Dynamically

- The `new` operator is used to allocate memory from the heap.
- Must specify type and number (for arrays) of objects.
- A pointer to the memory allocated is returned (if successful)
- If unsuccessful (e.g. out of memory), an **exception** is generated ("crashes", for now)
- Syntax:

```
<type> *p;
p = new <type>;      // allocate object, default constructor
p = new <type>(10);  // call constructor to initialized
p = new <type>[n];   // allocate array of size n
                     // default constructor called
```

- Allocated memory is uninitialized for basic types

Deallocating Memory

- Unlike local variables, memory allocated dynamically is used until explicitly deallocated.
- Should deallocate as soon as the memory is no longer needed—other parts of the program (or other programs) can re-use it
- Syntax:

```
delete p;              // deallocate single object
delete[] p;            // deallocate array
```

- Deleting a null pointer does nothing.
- Each `delete` must have a matching `new`. Deleting a pointer twice is an error.

Common Errors

Dereferencing uninitialized or null pointer: an uninitialized pointer can point anywhere!

Dangling pointer: two pointers to the same location, which has been returned to the heap by deleting one of the pointers. The other pointer is no longer valid.

```
int *p1 = new int;
int *p2 = p1;
delete p1;
*p2 = 10;                  // error
```

Memory leak: not returning memory, or losing reference to it.

```
int *p1 = new int;
p1 = new int;              // the address to old
                           // location is gone!
```

Common Errors

- Errors with pointers are hard to debug.
- A stray pointer may point to another variable—it gets overwritten instead of a run-time error.
- A stray pointer may also point to important "system information". Overwriting this can result in unpredictable behaviour.
- Memory leak will not result in a run-time error unless you run out of memory.
- Programs with errors may work on some machines but not others.

Example: Dynamic Array

```
int n;
cin >> n;
int *A = new int[n];   // allocate n elements
 ...
// change size to 2n
int *temp = A;         // don't lose the old one!
A = new int[2*n];
for (int i = 0; i < n; i++) {  // copy old elements
  A[i] = temp[i];
}
delete[] temp;         // delete the old array
temp = nullptr;        // ensure no dangling pointer
```

Example: 2-dimensional Dynamic Array

```
int **A;
A = new int *[m];    // first dimension
for (int i = 0; i < m; i++) {
  A[i] = new int[n];
}

for (int i = 0; i < m; i++) {
  delete[] A[i];        // delete the inner dimensions first
}
delete[] A;
```

References

- A reference is just an alias to the same variable. It must be initialized when declared.

```
int a = 10;
int &b = a;
b = 20;        // changes a too
```

- There is no need to dereference a reference.
- It cannot be changed once assigned. i.e. It can be treated as a constant pointer.
- Mostly used for parameter passing.

Pointers to Functions

- Pointer to a function: points to the address where the code for the function is stored.
- A function name is the starting address of its code.
- Definition:

```
returnType (*ptrName)(<parameterTypes>);
```

- A pointer to a function can be used to call the function it points to.

Pointers to Functions

```
int (*calc)(int x);
bool (*compare)(int A, int B);

int sqr(int n) { return n*n; }
int cube(int n) { return n*n*n; }
calc = sqr;
calc = cube;

bool same(int x, int y) { return x == y; }
compare = same;
```

Pointers to Functions

Usually passed as a parameter to function

```
void transform(int A[], int n, int (*f)(int) ) {
  for (int i = 0; i < n; i++) {
    A[i] = f(A[i]);
  }
}

transform(A, n, cube);  // cube every element
```

Type Conversions

- Automatic
  - arithmetic: narrow to wide is okay; wide to narrow may lose precision/overflow
  - bool: 0 is false, nonzero is true
  - arrays: array name is converted to constant pointer to first element
  - functions: function name converted to pointer to function
  - `nullptr`: can be automatically converted to `bool`
- Explicit
  - use `static_cast`
  - avoid old style cast: i.e. `(double)x`.

Miscellaneous

- (C++11) `array` class: acts like an array but provide some convenience features such as copying and comparing arrays.
- (C++11) "smart pointers": prevent aliasing, dangling pointers, memory leak. There are many kinds to choose from.
- (C++11) `auto` variable type for complicated data types.

Vectors

- It is often convenient to have an array that grows automatically when needed.
- A vector is exactly what we need. It is a container and can contain any type (but all objects must have the same type).
- Syntax:

```
#include <vector>
using namespace std;

vector<int> A;          // a vector of int
vector<char *> B;       // a vector of char *
vector<double> C(3);  // a vector of 3 doubles (0.0)
vector<int> primes { 2, 3, 5, 7, 11, 13, 17 };  // C++11
```

Vectors

- Other constructors: see references
- To get the size of vector: `A.size()`
- ith element: `A[i]`, as long as i is between 0 and size-1. e.g.

```
for (int i = 0; i < A.size(); i++)
  cout << A[i] << endl;
```

- Can assign or compare a vector:

```
bool t = (v1 == v2);  // true if all elements of v1 and v2
                      //   are the same, and
                      //   v1.size() == v2.size()
v1 = v2;              // each member of v2 is copied to v1
```

Vectors

- To grow a vector, use `push_back()`:

```
vector<int> v;
v.push_back(4); v.push_back(3); v.push_back(2);
                          // v has elements [4, 3, 2]
```

- Use `pop_back()` to shrink a vector.
- See any C++ reference web site for other functions.