ALGORITHMS AND DATA STRUCTURES

POINTERS AND ARRAYS

C++ MEMORY MODEL

- How information is stored in a computer?
  - Information is stored as combinations of *bits*
    - 1 bit [b] = contraction(“binary digit”)
      - takes two values: 0/1; off/on; false/true
    - 1 byte [B] = 8 bits – 1B = 8b [= `sizeof(char)`]
    - 1 word [w] = `sizeof(int)` [= 4B or 8B], defined like that on most machines

BINARY AND HEXADECIMAL REPRESENTATIONS

- Bits in an unsigned integer are encoded using *binary notation*, that is, 0's and 1's
  - What about signed `int`'s? ***Two's complement arithmetic***
- Booleans seem to be naturally represented by one single bit. Is this the case in modern computers?
- Hexadecimal notation is mostly used to refer to memory addresses in the RAM

C++ MEMORY LAYOUT

- Memory is usually specified in terms of bytes
- Memory is divided into three main regions. Can you name them?

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

C++ MEMORY LAYOUT

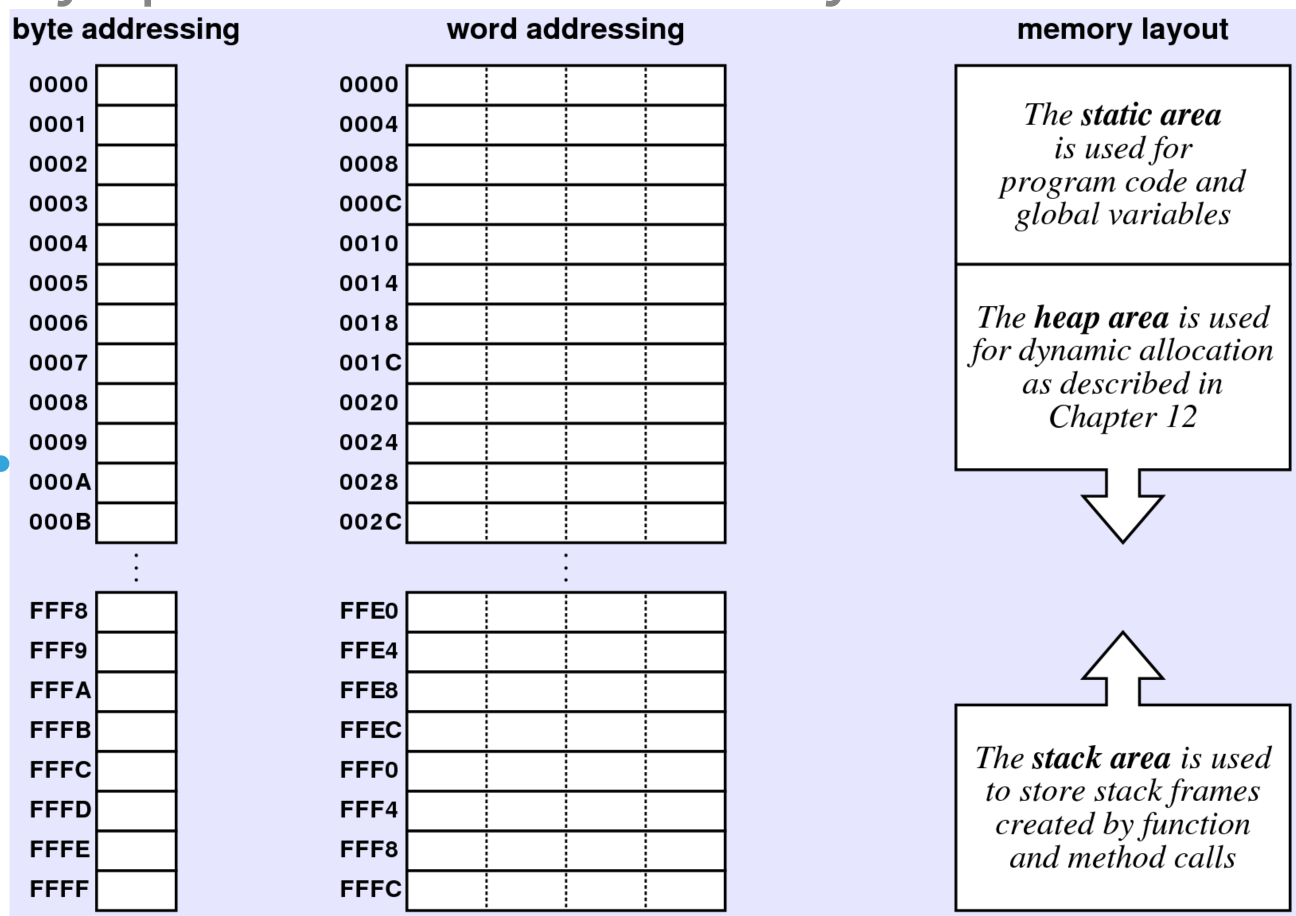
- Memory is usually specified in terms of bytes
- Memory is divided into three main regions. Can you name them?
  - Static area: Code and global and static variables
  - Heap area: Dynamic allocation
  - Stack area: Stack frames, local variables

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

C++ MEMORY LAYOUT

- Memory is usually specified in terms of bytes

Memory diagrams

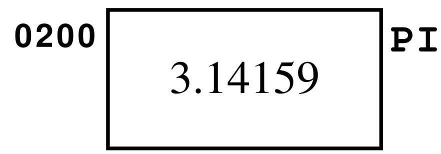


Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

ASSIGNING MEMORY TO VARIABLES

- Where is this variable stored?

```
const double PI = 3.15159;
```



- What about this one?

```
map<string, bool>::const_iterator it = mp.cbegin();
```

- And this other one?

```
Complex pt = new Complex(3, 4);
```

EXAMPLE: POWERS OF TWO

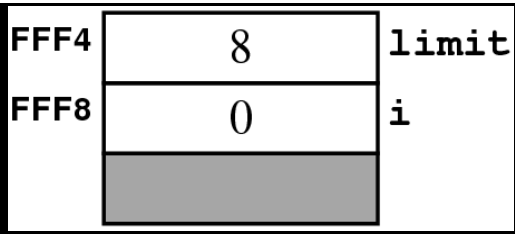
```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```

EXAMPLE: POWERS OF TWO — MAIN

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```

EXAMPLE: POWERS OF TWO — MAIN

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```



EXAMPLE: POWERS OF TWO — TWO POWER

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```

EXAMPLE: POWERS OF TWO — TWO POWER

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```

FFE0		result
FFE4		i
FFE8	0	k
FFEC	2	n
FFF4	8	limit
FFF8	0	i

POINTERS

- hold the address in memory of a variable
- are data items whose value is an address in memory
- have the same data type as the variable they have address
- allow you to refer to a large data structure in a compact way
- make it possible to manage and reserve memory during program execution
- can be used to record relationships among data items

DECLARING POINTERS VARIABLES

- To declare a variable as a pointer use the asterisk '*'

```
int * pointer;
char * cptr;
```

- declares `pointer` and `cptr` to be types pointer-to-`int` and pointer-to-`char`
- Syntactically, the '*' belongs to the variable not the type
- What is the difference between these two statements?

```
double *ptr1, *ptr2;
double *ptr1, ptr2;
```

DECLARING POINTERS VARIABLES

- To declare a variable as a pointer use the asterisk '*'

```
int * pointer;
char * cptr;
```

- declares `pointer` and `cptr` to be types pointer-to-`int`

are distinct types, though they are represented as addresses

- Syntactically, the '*' belongs to the variable not the type
- What is the difference between these two statements?

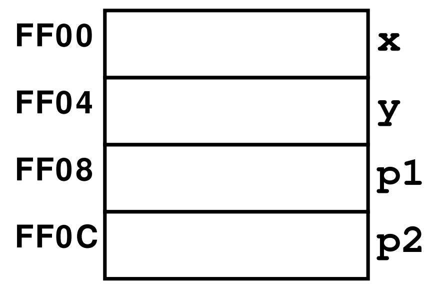
```
double *ptr1, *ptr2;
double *ptr1, ptr2;
```

ADDRESS-OF AND DEREFERENCING OPERATORS

- **‘&’**: **address-of**
  - returns the memory address in which the value was stored
- **‘*’**: **value-pointed-to**
  - returns the value to which the pointer points to
- To understand these operators, consider the next example

EXAMPLE: POINTERS

```
int x, y;
int *p1, *p2;
```



Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

EXAMPLE: POINTERS

```
int x, y;
int *p1, *p2;

x = 42;
y = 163;
```

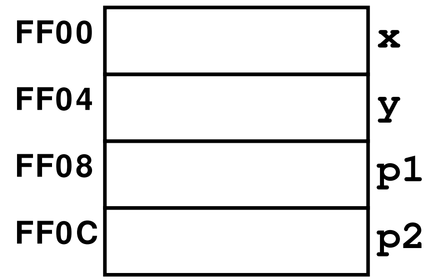
FF00		x
FF04		y
FF08		p1
FF0C		p2

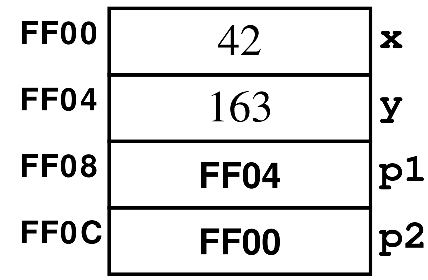
FF00	42	x
FF04	163	y
FF08		p1
FF0C		p2

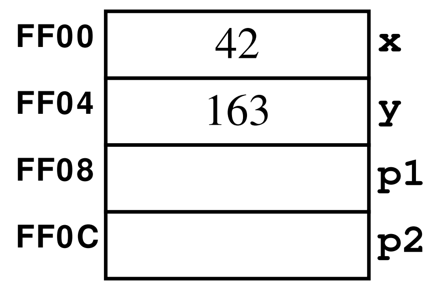
Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

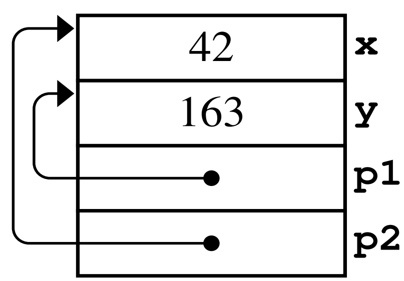
EXAMPLE: POINTERS

```
int x, y;
int *p1, *p2;

x = 42;
y = 163;

p1 = &y;
p2 = &x;
```





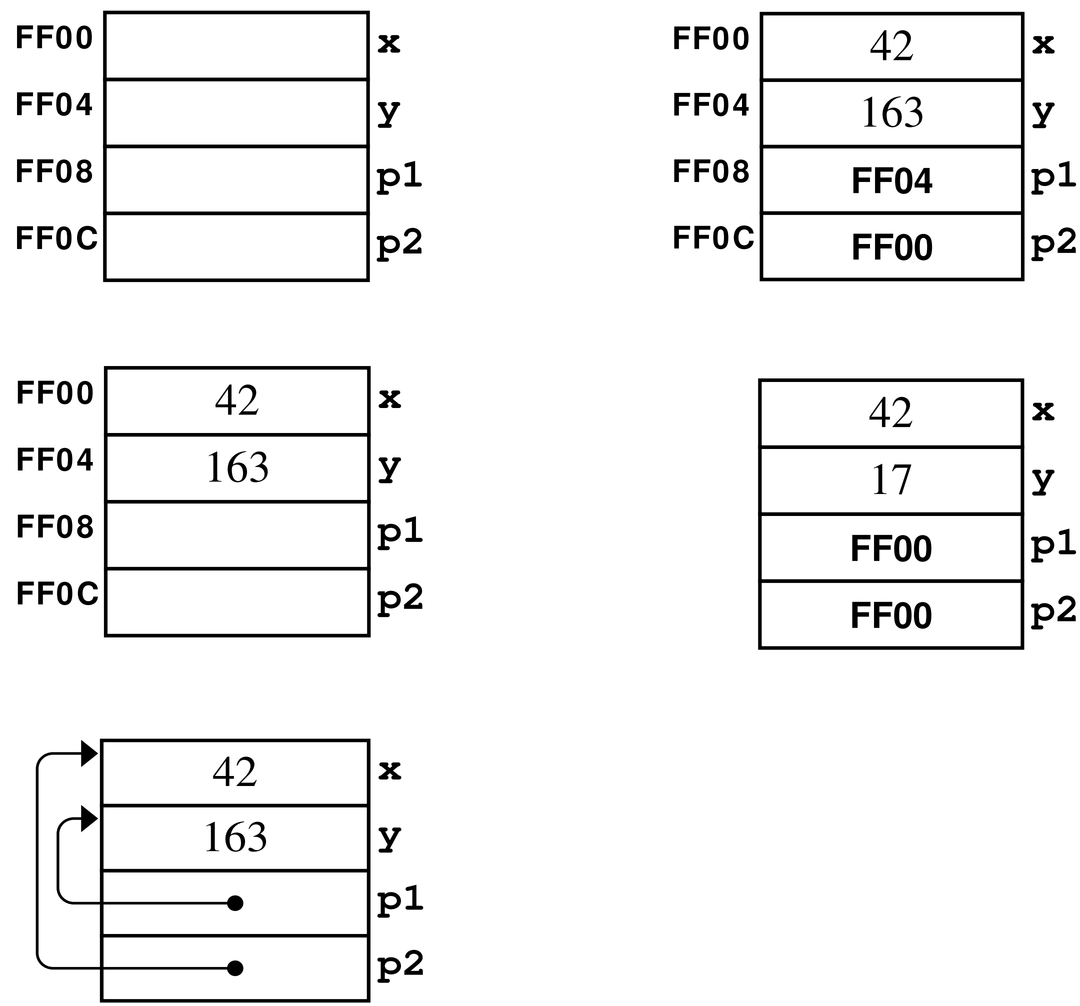




Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

EXAMPLE: POINTERS

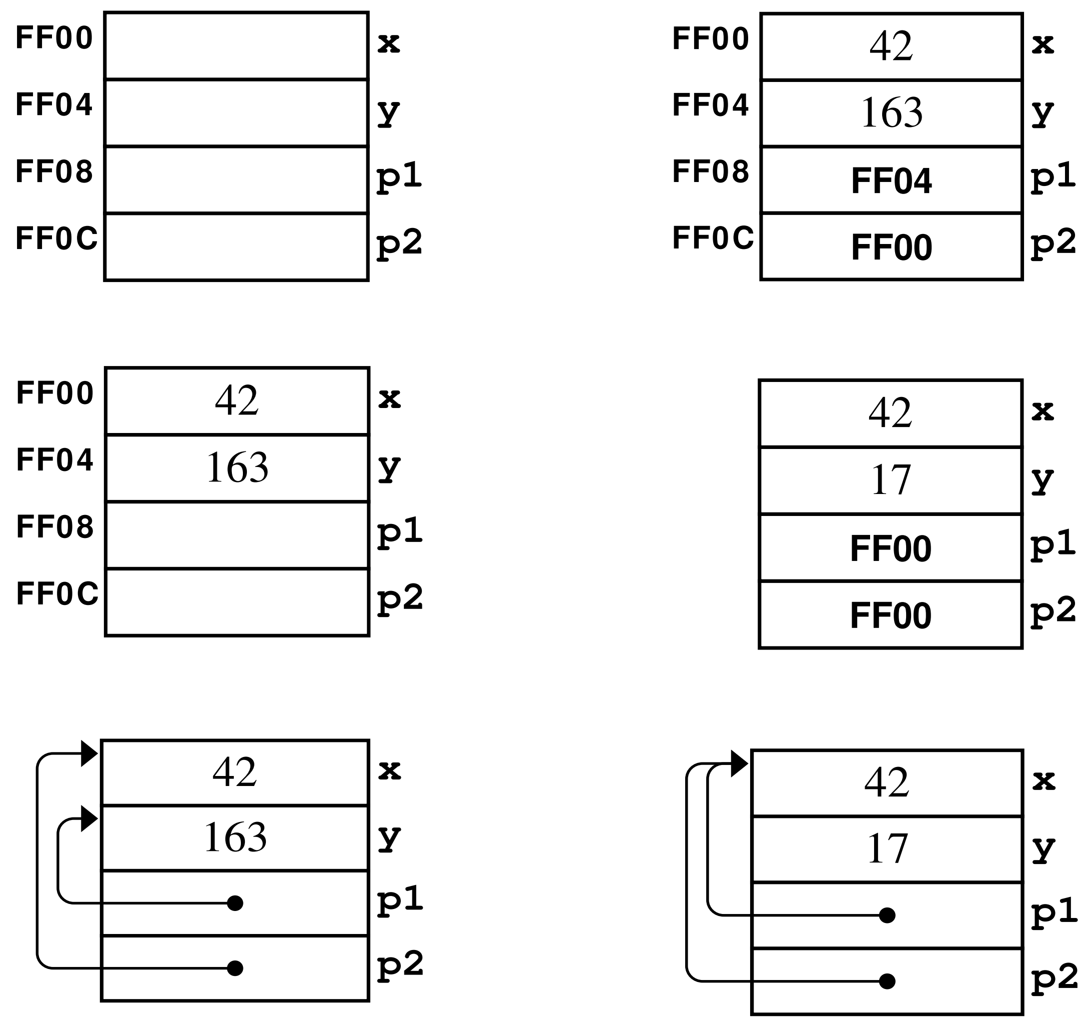
```
int x, y;
int *p1, *p2;

x = 42;
y = 163;

p1 = &y;
p2 = &x;

*p1 = 17;
```



Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

EXAMPLE: POINTERS

```
int x, y;
int *p1, *p2;

x = 42;
y = 163;

p1 = &y;
p2 = &x;

*p1 = 17;

p1 = p2;
```

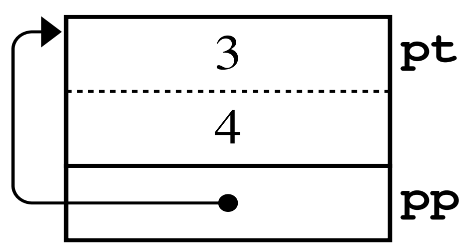


Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

POINTERS TO STRUCTURES AND OBJECTS, AND 'THIS'

- Two local variables:
  - `pt` contains a `Point` object with the coordinate values 3 and 4
  - `pp` contains a pointer to that same `Point` object

```
Point pt(3, 4);
Point *pp = &pt;
```



POINTERS TO STRUCTURES AND OBJECTS, AND ‘THIS’

- Two local variables:
  - `pt` contains a `Point` object with coordinate values (3, 4)
  - `pp` contains a pointer to that same `Point` object
- **`this`**: pointer to the current object
  - use to select instance variables

```
Point(int cx, int cy) {
    x = cx;
    y = cy;
}

Point(int x, int y) {
    this->x = x;
    this->y = y;
}
```

POINTERS TO STRUCTURES AND OBJECTS, AND 'THIS'

- How to invoke methods and fields?
  - Wrong, why? → `*pp.getX();`
  - What is this? → `*(pp.getX());`
  - And that? → `(*pp).getX();`
  - Wow, wait! → `pp->getX();`
- How to tell what is going on?

Operators organized into precedence groups	Associativity
`()` `[]` `->` `.`	*left*
unary operators: `-` `++` `--` `!` `&` `*` `~` `(`*type*`)` `sizeof`	*right*
`*` `/` `%`	*left*
`+` `-`	*left*
`<<` `>>`	*left*
`<` `<=` `>` `>=`	*left*
`==` `!=`	*left*
`&`	*left*
`^`	*left*
`|`	*left*
`&&`	*left*
`||`	*left*
`?:`	*right*
`=` *op*`=`	*right*

POINTERS TO STRUCTURES AND OBJECTS, AND 'THIS'

- How to invoke methods and fields?
  - Wrong, why?
  - What is this?
  - And that?
  - Wow, wait!

```
*pp.getX();

*(pp.getX());

(*pp).getX();

pp->getX();
```

- **'->'**: **arrow operator**
  - combines dereference and selection into a single operator

THE NULL POINTER [OLD!]

- Pointer value that indicates that the pointer does **not** in fact refer to any valid memory address
- It is illegal to use the dereferencing operator (*) on `NULL`
- Keyword: `NULL`
- Defined in the interface `<cstddef>`

```
Point * pointer_to_pt;
// some action done on pointer_to_pt
if (pointer_to_pt == NULL)
    cerr << "Somethin's NOT right. Check!"
```

THE NULLPTR POINTER [NEW?]

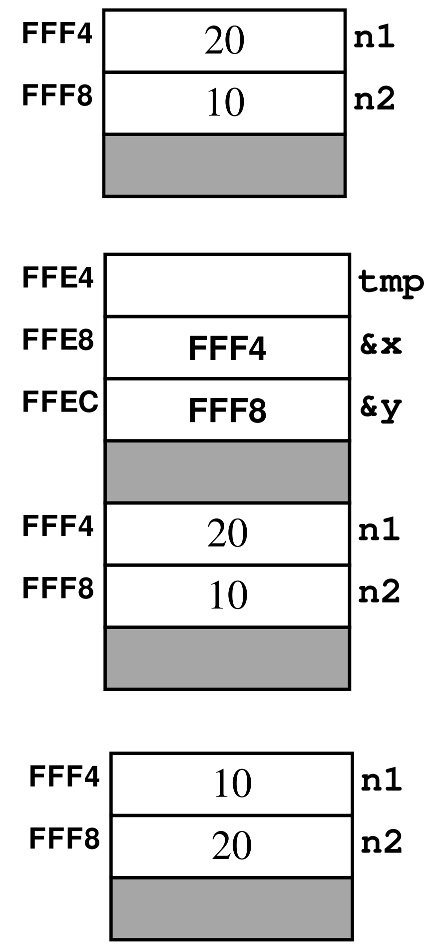
- A null pointer does not point to any object
- Code can check whether a pointer is null before attempting to use it
- **nullptr** is a literal that has a special type that can be converted to any other pointer type

```
int *p1 = nullptr; // equivalent to int *p1 = 0;
// must #include <cstdlib>
int *p3 = NULL; // equivalent to int *p3 = 0;

/* avoid using NULL, modern C++ should use nullptr */
```

PASSING BY REFERENCE

- The stack frame stores a pointer to the location in the caller at which that value resides

```
int main() {
    int n1 = 20, n2 = 10;
    if (n1 > n2) swap(n1, n2);
    cout << n1 << " " << n2 << endl;
    return 0;
}

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



Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

PASSING BY REFERENCE

- Any changes are made to the target of the pointer, which means changes remain in effect after the function returns
- Equivalent implementation without address-of operator

```
void swap(int *px, int *py) {
    int tmp = *px;
    *px = *py;
    *py = tmp;
}

// function should be called like this
swap(&n1, &n2);
```

ARRAYS

- are low-level collections of individual data values
- are countable (count collection size)
- are homogeneous (same data type)
- **Limitations**:
    - Have a fixed, unchangeable size
    - Offer no support for inserting/deleting elements
    - There is no bound-choking procedure

ARRAY DECLARATION

```
const int N_ELEMS = 1024;

// array declaration: type name[size];
int arrayI[10], intArray[N_ELEMS];

for (int i = 0; i < N_ELEMS; i++) {
    // array selection
    intArray[i] = 10 * i;
    cout << i << " " << intArray[i] << endl;
}

// static initialization
const int DIGITS[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

intArray

0	10	20	30	40	50	60	70	80	90
0	1	2	3	4	5	6	7	8	9

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

POINTERS AND ARRAYS

- Array is synonymous with a pointer to its initial element

```
void sort(int array[], int n) {
    for (int lh = 0; lh < n; lh++) {
        int rh = lh;

        for (int i = lh + 1; i < n; i++)
            if (array[i] < array[rh]) rh = i;

        swap(array[lh], array[rh]);
    }
}

void sort(int *array, int n) {
    // ...
}
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

POINTER ARITHMETIC

- Operators ‘+’ and ‘–’ can be applied to pointers

```
int array[10];
int *p, *q, k = 6;

// p and q point to the 1st address of array
p = array;
q = &array[0];

// p1 and p2 are equivalent
int *p1;
p1 = p + k;
int *p2 = &array[k];

// increment and decrement of pointers
*p1--; // how to check what is this doing?
*p1++; // and that
```

ALGORITHMS AND DATA STRUCTURES

DYNAMIC MEMORY ALLOCATION

STYLES OF MEMORY ALLOCATION

- There are three basic styles of allocation
  - **Static**: Global variables

STYLES OF MEMORY ALLOCATION

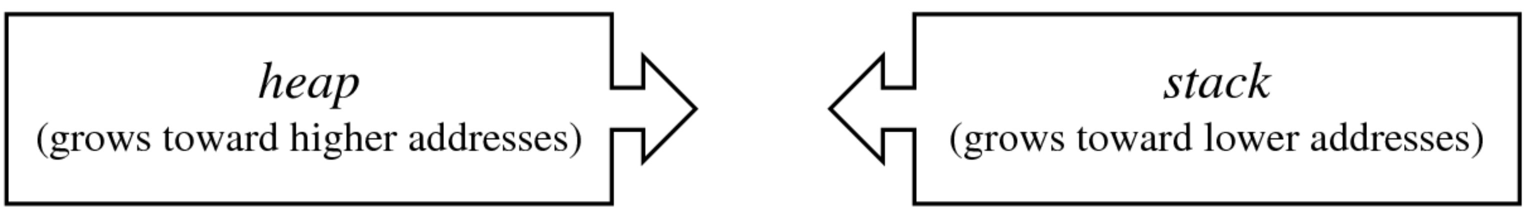
- There are three basic styles of allocation
  - **Static**: Global variables
  - **Automatic**: Local variables inside function

STYLES OF MEMORY ALLOCATION

- There are three basic styles of allocation
  - **Static**: Global variables
  - **Automatic**: Local variables inside function
  - **Dynamic**: Require memory space as program runs
    - Allocate and free memory
    - Takes place in the heap (pool of available memory)

DYNAMIC ALLOCATION AND THE HEAP

- C++ allows to allocate some of the unused storage to the program whenever your application needs more memory
- **Example**: If you need an array while the program is running, you can reserve part of the unallocated memory, leaving the rest for subsequent allocations
- The pool of unallocated memory available to a program is called the **heap**



THE NEW OPERATOR

- is the way to allocate memory from the heap
- returns the address of a storage location in the heap
- once allocated in the heap, one can refer to that variable

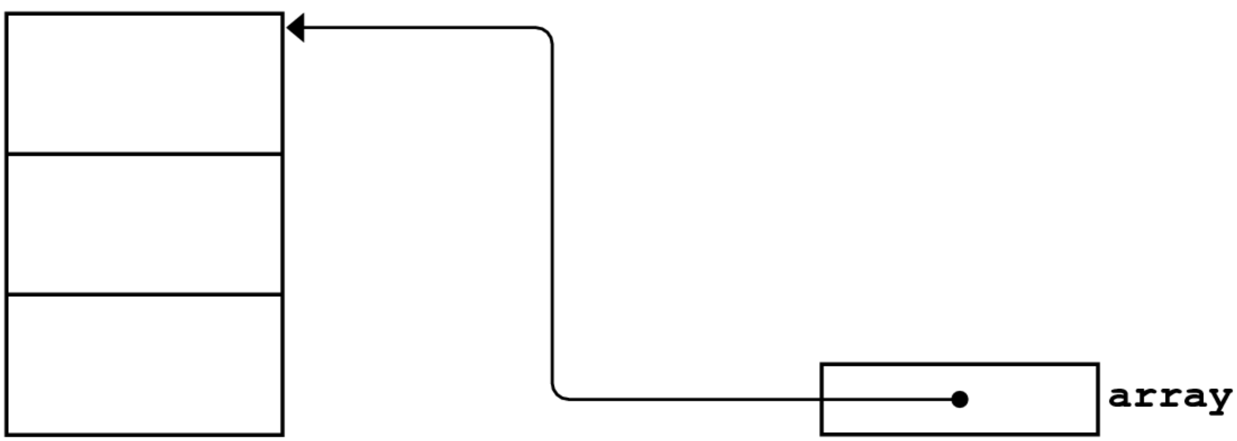
```
int *ip = new int;
*ip = 42;
```

- can be used to allocate objects and *dynamic arrays* on the heap

DYNAMIC ARRAYS

- is an array allocated in the heap
- To allocate them use the following syntax

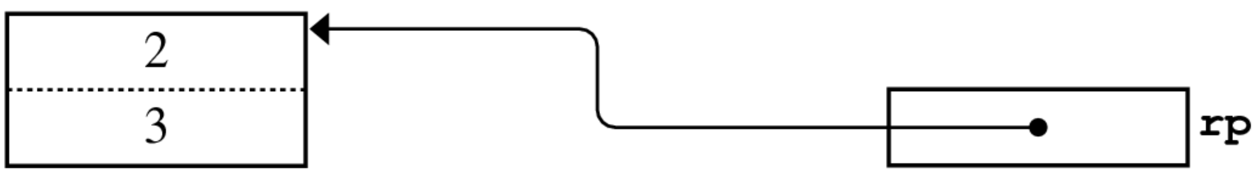
```
double *array = new double[3];
a[0] = a[1] = a[2] = 0.0;
```



DYNAMIC OBJECTS

- With new can allocate general structures like objects
- To allocate a `class` use the following syntax

```
Rational *rp = new Rational;
Rational *rq = new Rational(2, 3);
cout << *rq << endl;
```



- If supply arguments after the type name, C++ will call the matching version of the constructor

THE DELETE OPERATOR

- takes a pointer allocated by new and frees the memory associated with that pointer
- If the heap memory is an array, you need to add square brackets after the delete
- Some languages support automatic memory freeing that is no longer in active use (***garbage collection***)

```
delete ip; // free memory occupied
delete[] array; // free allocated memory
```

DESTRUCTORS

- C++ does not have garbage collector
- But, each class is allowed to specify what happens when an object of that class disappears
  - each class takes responsibility for its own heap storage
- Each class can define a ***destructor***, which is called automatically when an object of that class disappears
  - Destructors can perform a variety of cleanup operations

DESTRUCTORS

- cannot be overloaded
- are declared with a tilde '~' character
- do not have a return type like constructors
- **Example 1**: for a class `Rational` the destructor's prototype looks like `~Rational();`.
- **Example 2**: The `CharStack` class has a destructor prototyped as `~CharStack();`.