Structures

Data aggregates

Like classes (in Java) except usually contain no func

Structure members are public (we'll learn this later)

```
struct student_data
{
  char      name[30];
  int       age;
  int       sid;
}; /* <== DO NOT FORGET the semicolon */
```

typedef

typedef <type definition> new_type_name;

```
typedef struct student_data
{
  char      name[30];
  int       age;
  int       sid;
} student;
```

Structures

```
#include <iostream>

typedef struct student_data
{
  char     name[30];
  …
} student;

int main(int argc, char* argv[]) {
  student amza; //or struct student_data amza;
  cin >> amza.name;
  cin >> amza.age;
  cin >> amza.sid;
  cout << "name =" << amza.name << ", age =" << amza.age << endl;
  return 0;
}
```

Pointers

Topics

- Simple memory allocation and addressing
  - Pointers
  - Example Mechanisms
  - Operators for use with pointers
- Dynamic memory allocation

Harsh Reality

Memory Matters

Memory is not unbounded

- It must be allocated and managed

Memory referencing bugs especially pernicious

- Effects are distant in both time and space (e.g., accessing an uninitialized variable).

Memory Management

A variable lives in some memory location for some time

Memory is allocated to variables in two ways

- Automatic allocation: through variable declaration
  - e.g., int i; //allocates 4 bytes for i upon entering scope/function
- Dynamic allocation: using new
  - e.g., new double; //allocates 8 bytes upon calling new

Scope = Enclosing block for a variable - could be a func

or

artificially created by using {int i; ……} within func body

Memory Management

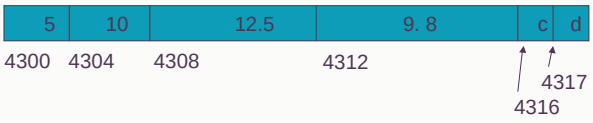
A variable lives in some memory location for some time

The variable lifetime is

- Automatic allocation: within scope
  - e.g., int i; //while enclosing function instantiation is active
- Dynamic allocation: until programmer explicitly frees block
  - e.g., new double; //until program calls delete (outlives scope)

Memory layout and addresses

```
Example using automatic allocation:

int x = 5, y = 10; //automatic vars
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```



Pointers

Definitions:

"Pointers are variables that hold a memory address"

e.g., a pointer p contains an address addr

The memory address addr contains another variable var

We say that pointer p "points to" variable var

Pointers

Definitions:

"Pointers are variables that hold a memory address"

We say that pointer p "points to" variable var

Declarations:

```
float f;      //variable of type float
float *p;   //pointer to variable of type float
```

Pointer Initialization/Assignment

Q: How do we get the memory address of a variable ?

A: the "get address" operator: &

```
float f;      //variable of type float
float *p;   //pointer to variable of type float
p = &f;
```

Data Representations (revisited)

Sizes of C++ Objects (in Bytes)

Data Type	Compaq Alpha	Typical	Intel IA32
int	4	4	4
long int	8	4	4
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
char *	8	4	4

» Or any other pointer

"Word Size" (Convention)

- Size of integer data (i.e., typically 4 bytes)

Pointer Dereferencing

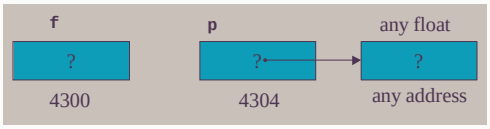
Q: Get the value of the variable “pointed-to” by pointer

A: the “indirection” operator: *

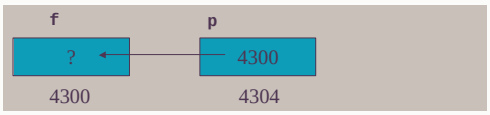
```
float f = 3.2;      //variable of type float
float *p;           //pointer to variable of type float
p = &f;
cout << *p;         //prints the value of var “pointed-to” by p
                    // (i.e., var at address p)
```

Using Pointers (1)

```
float f;        /* data variable */
float *p;       /* pointer variable */
```




```
p = &f;     /* & = address operator */
```



Pointer Dereferencing

Q: Get the value of the variable “pointed-to” by pointer

A: the “indirection” (a.k.a. “dereferencing”) operator: *

```
float f;        //variable of type float
float *p;       //pointer to variable of type float
*p = 3.2;       //WRONG !!
                //Dereferencing an unitialized pointer
                //Typically results in SEGFAULT (bombing)
```

Pointer Dereferencing

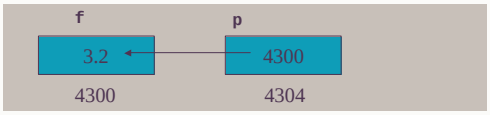
Q: Get the value of the variable “pointed-to” by pointer

A: the “indirection” operator: *

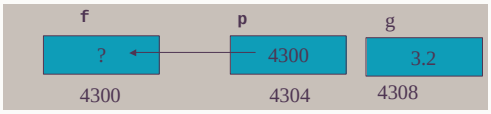
```
float f;        //variable of type float
float *p = &f;  //pointer to variable of type float
*p = 3.2;       //LHS is the var “pointed-to” by p
cout << f;      //prints the value of var
```

Pointers made easy (2)

```
float f;        /* data variable */
float *p;       /* pointer variable */
*p = 3.2;       /* use of indirection operator */
```

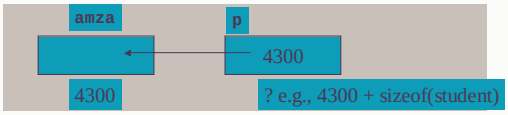



```
float g = *p;       /* indirection: g is now 3.2 */
*p = 1.3;           /* f becomes 1.3 but g is still 3.2 */
```



Pointers made easy (2)

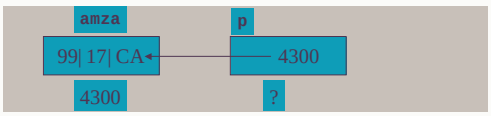
```
student amza;     /* data variable */
student* p;       /* pointer variable */
p = &amza;        /* use of indirection operator */
```




```
(*p).sid = 99;       /* indirection: amza.sid is now 999*/
(*p).age = 17;     /* amza.age is 17 */
cin >> (p*).name;  /* say we input "CA" */
```

The → Operator (instead of * and .)

```
student amza;     /* data variable */
student* p;       /* pointer variable */
p = &amza;        /* use of indirection operator */

p → sid = 99;       /* indirection: amza.sid is now 999*/
p → age = 17;     /* amza.age is 17 */
cin >> p → name;  /* say we input "CA" */
```



Dynamic Memory Allocation

Java manages memory for you, C++ does not

- In Java programmer allocates with new, but does not free space (garbage collection in Java)
- C++ requires the programmer to *explicitly* allocate and deallocate memory
- Memory can be allocated dynamically during run-time with `new` and deallocated (freed) using `delete`

Memory

Memory allocated with new, de-allocated with delete

new returns address of (pointer to) allocated block

The memory allocator provides an abstraction of memory as a set of blocks

Use of New/Delete

`new int;`

- If successful:
  - Returns a pointer to a memory block of at least `sizeof(double)` bytes, i.e. 8, (typically) aligned to 8-byte boundary.

`delete p;`

- Returns the block pointed to by p to pool of available memory
- p must come from a previous call to new.

new

Allocates memory in the heap

- Lives between function invocations

Examples

- Allocate an integer
  - `int* iptr = new int;`
- Allocate a structure
  - `struct student_data* amza = new student;`
    `(same as: student* amza = new student;)`
    `(same as:`
    `student* amza = new struct student_data;)`

delete

Deallocates memory in heap.

Pass in a pointer that was returned by new.

Examples

- Allocate an integer
  - `int* iptr = new int;`
  - `delete iptr;`
- Allocate a structure
  - `struct student_data* amzaptr = new student;`
  - `delete amzaptr;`

Caveat: don't free the same memory block twice!

Examples

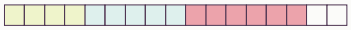
```
typedef struct student_data
{
  char     name[30];  …
} student;


typedef struct two_chars {
  char     first_char;
  char     second_char;
} two;

typedef struct five_chars {
 char     first_char;
 char     second_char; …
… char     fifth_char;
} five;
```

Allocation/Dealloc. Examples

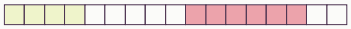
```
p1 = new int;
```

```
p2 = new struct five_chars;
```

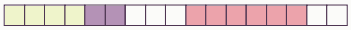
```
p3 = new struct six_chars;
```



```
delete p2;
```



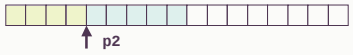
```
p4 = new struct two_chars;
```



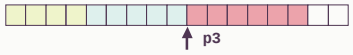
Allocation/Dealloc. Examples

```
p1 = new int;
```

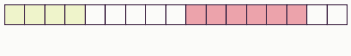

```
p2 = new five;
```



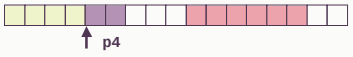

```
p3 = new six;
```




```
delete p2;
```



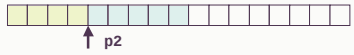
```
p4 = new two;
```



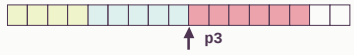
Pointer Declarations

```
int *p1; five *p2; six *p3; two *p4;
p1 = new int;
```

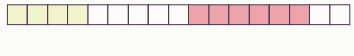

```
p2 = new five;
```



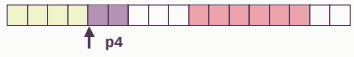

```
p3 = new six;
```




```
delete p2;
```



```
p4 = new two;
```



Dynamic Allocation Example

```
void foo(int n, int m) {
  int i, *p;  //automatic allocation

  /* dynamically allocate a block of 4 bytes */
  if ((p = new int) == NULL) {
    cerr << "allocation failed";
    exit(0);
  }

  *p = 5;

  /* print the content of the newly allocated space */
  cout << *p << endl;

  i = *p;

/* print the content of i */
  cout << i << endl;

  delete p; /* return 4 bytes to available memory */
            /* cannot access this space with *p anymore */

/* print the content of i */
  cout << i << endl;
}
```

How about pointers inside structs ?

```
typedef struct four_chars {
 char     first_char;
 char     second_char; …
… char     fourth_char;
} four;

typedef struct four_plus_two_chars {
  four   *first_four_chars;
  two    *last_two_chars;
} four_plus_two;
```

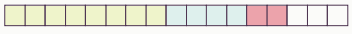
How about pointers inside structs ?

Need to allocate nested objects. Need to deallocate them correspondingly.

```
p1 = new four_plus_two_chars;
```

```
p2 = new four;
```

```
p3 = new two;
```



```
delete p1;
```

Oops !

Memory leak ! (garbage left around)

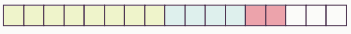
How about pointers inside structs ?

Need to allocate nested objects. Need to deallocate them correspondingly.

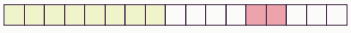
```
p1 = new four_plus_two_chars;
```

```
p2 = new four;
```

```
p3 = new two;
```



```
delete p1->first_four_chars;
```



```
delete p1->last_two_chars;
```

```
delete p1;
```

Summary: Not like Java

No garbage collection

Operator **new** is still a high-level request such as
"I'd like an instance of class `String`"

Try to think about it low level

- You ask for *n* bytes (the sizeof that type/class)
- You get a pointer (memory address) to the allocated object

Process Memory Image

Automatic variables are allocated memory on the stack.
Stack grows downwards

Dynamic Memory Allocator requests memory from the heap.
Heap grows upwards