CSIS/COMP 1117B
Computer Programming

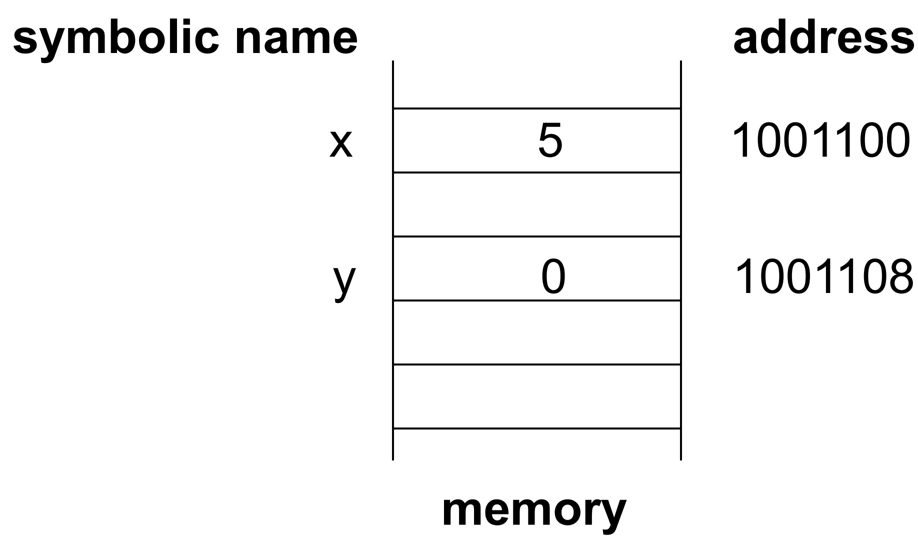
Pointers, Dynamic Arrays and Dynamic Structures

Pointers

- Recall that a **variable** is a sequence of memory cells for storing a value and that a variable is typically identified by a (symbolic) **name**.
- A **pointer** is the memory address of a variable.
- For example:

  int x = 5;

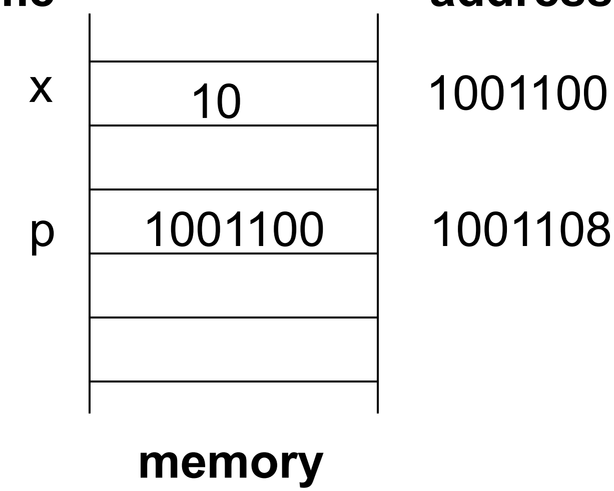
  int y(0);



Pointer Variables

- A **pointer variable** is a variable having pointers as value.
- For example:

```
int x = 10;
int *p;  // p is a pointer to int
. . .
// p gets address of x
p = &x;
// prints value of variable pointed at by p
cout << “x [“ << *p << “]\n”;
```



Declaration of Pointer Variables

- A pointer declaration starts with the type of value being ***pointed at***, and each name of the variables is prefixed with an asterisk (*)
- For example:

```
char *t;     // t is a char pointer
int *r, *q;  // r and q are int pointers
// d is a double pointer and is initialized to NULL
double *d = NULL;
```

- **NULL** is a special pointer constant interpreted as ***pointing nowhere***.
- The definition of NULL is in <cstddef>

Pointer Operations

- **Assignment** (=): assigns a pointer value (address) to a pointer variable; for example:
  t = NULL; // set t to point at nowhere
  r = q;   // copies value of q to r
- **Address of** (&): gets the address of a variable, for example:
  &d evaluates to the ***address of the pointer variable*** d
  &y evaluates to the ***address of the int variable*** y
- **Dereference** (*): refers to the variable pointed at by a pointer variable, for example:
  d** is the ***double variable pointed at by d
  *t is the ***char variable*** pointed at by t
- **Comparison** (==, !=): checks whether or not two pointer variables contain the ***same value***, for example:
  if (p == NULL) . . .
  while (q != r) . . .

Type Compatibility

- The **type** of a pointer variable depends on the type of value it points to.
- Assigning a pointer value to a pointer of a different type is not allowed, for example, the following assignments are ***illegal***

  t = d; // cannot assign a double pointer to a char pointer

  q = &x; // cannot assign a int pointer to a double pointer

  x = p; // cannot assign a int pointer to an int
- Type casting can be used to change the type of a pointer variable if necessary, for example:

  t = (char *) d;

  x = (int) p; // address typically size of an int, no conversion

Array and Pointer

- An array variable ***stores*** the ***address*** of the first array element, i.e., an array variable by itself is a pointer variable (to its element type).
- An array name can be used as a ***constant*** pointer value.
  - ISO C++ forbids casting to array types so that the following assignments are ***illegal***:
    ```
    double a, b[] = {1.0, 2.0 };
    b = &a;
    b = (double [ ])&a;
    ```
  - ISO C++ forbids assignments to array variables, the above assignments are ***illegal*** irrespective of what is on the right hand side!

Pointer Arithmetic

- The following functions generate the same output:

```
void print_array(double a[ ], int n) {
    for (int i = 0; i < n; i++) {
        //  *a delivers the value of the variable pointed at by a
        //  a++ advances a to point at the next array element
        //  a + 1  ->  a + size of object pointed at by a
        cout << ‘ ‘ << *a++;
    }
    cout << endl;
}
void print_ptr(double *a, int n) {
    for (int i = 0; i < n; i++) {
        cout << ‘ ‘ << a[i];    //  a[i] is equivalent to a + i
    }
    cout << endl;
}
```

Array as Parameter

- When an array is passed as parameter to a function, the elements of the array are ***not copied*** to the corresponding parameter of the function; instead, the array name is treated as a ***pointer variable*** and the address of the first element of the array is passed to the corresponding parameter.
  - ++ or – – is applicable to an array name only when the array name is actually a pointer variable
- **The values of the elements of an array passed as parameter (by value) to a function could be modified in the body of the function as if the array elements had been *passed by reference*!**

Pointer as Parameter

- Similar to the situation with elements of an array, when a pointer value is passed as a value parameter to a function, the variable pointed at by the pointer value can be modified by the body of the function; it is as if the variable pointed at by a pointer value had been passed by reference to a function.
- For example swap can be re-written as:

```
// C version of swap (C only has by value parameters)
void swap(int *a, int *b) {
    int t = *a;
    *a = *b; *b = t;
}
. . .
int x(10), y(20);
// parameters to swap are pointers
swap(&x, &y);
```

Dynamic Arrays

- A **dynamic array** is an array that is created using the **new** operator during execution.
- Dynamic arrays allows more efficient use of the available storage as large arrays are only allocated when needed and can be freed by using the **delete** operator as soon as they are not needed.
- Dynamically allocated storage is managed by the ***system*** and is separated from the data area of a program.

Declaration and Creation

- Any pointer can be used to represent a dynamic array of values *appropriate* to that pointer, for example:

  ```
  int *a;    // a can be used as a dynamic int array
  char *p; // p can be used as a dynamic char array
  ```

- A dynamic array is created using the new operator which is of the form:

  <pointer variable> ‘=‘ new <type> ‘[‘ <array size> ‘]’ ‘;’

- For example:

  ```
  // a points at a dynamic int array of size 1000000
  a = new int[1000000];
  // p points at a dynamic char array of size 40000000
  p = new char[40000000];
  ```

Usage and Deletion

- Referencing individual elements of a dynamic array is exactly the same as a conventional array, for example:

  a[k+1] // $(k+2)^{th}$ element of a, an int variable

  p[300] // 301^{th} element of p, a char variable

- The storage occupied by a dynamic array can be returned to the system by using the **delete** operator, for example:

  delete [ ] a;

  delete [ ] p;

Dynamic Structures

- Similar to dynamic arrays, there are **dynamic structures**.
- A dynamic structure is accessed via a pointer to the structure, for example:

  student *u;   // u is pointer to student
- A dynamic structure is created using the new operator of the form:

  <struct pointer> ‘=‘ new <struct type> ‘;’
- For example:

  u = new student;   // u points at a new student struct
- When a dynamic structure is no longer needed, it can be freed using the delete operator:

  delete u;   // return storage of struct pointed at by

Referencing Members of Dynamic Structures

- Given u a pointer to a struct, *u is a struct and the dot (.) operator can be used with *u to access members of the struct, for example:

```
(*u).name        // name of student pointed at by u
(*u).uid         // uid of student pointed at by u
(*u).name.first  // first of name of student pointed at by u
```

- Since dynamic structures are quite common for programming data structures, there is a more convenient form for referencing members of a dynamic structure using the ***arrow*** operator (->):

```
u->name        // name of student pointed at by u
u->uid         // uid of student pointed at by u
u->name.first  // first of name of student pointed at by u
```

Dynamic Array of Dynamic Structure

```
// declare sdb a pointer to pointer to student
student **sdb
// allocate a dynamic array of 10 student pointers
sdb = new student*[10];
// sdb[0] points at a dynamically allocated student
sdb[0] = new student;
// initialize first of name of student pointed at by first element of sdb
sdb[0]->name.first = “John”;
// initialize uid of student pointed at by first element of sdb
sdb[0]->uid = 32768803;
// release student pointed at by first element of sdb
delete sdb[0];
// release array pointed at by sdb
delete [ ] sdb;
```

The Original Question

- Cannot use individual variables to implement individual accounts, ***why?***
  - need to organize accounts in certain ways
    - need to lookup a customer using acc_num
    - need to be able to add new accounts and delete old accounts
- Operation to create an account
  - obtain name and balance from user
  - generate a new account number
  - *is there space for one more account?*
- Operation to close an account
  - obtain number of account to be closed
  - check that there really is such an account and *close* it

Operator Precedence Again

high priority

resolution ::

dot ., ->, *index* [ ], *call* ( ), k++, k--

unary +, *unary* –, ++k, --k, !, (*type*), *dereference* *, *address of* &, new, delete, sizeof

*, /, %

+, –

>=, <=, >, <

==, !=

&&

||

?:

=

low priority

The `this` Pointer

- The `this` pointer is a **predefined** pointer that points to the calling object.
- It is useful for checking if a member method is being called with the object as one of its parameters, for example:

```
class & class::operator = (
const class & right_side) {
    if (this == right_side) return *this;
    else {
          // create a copy of right_side and
          //   copy to left_side
          return *this;
    }
}
```