ENGINEERING PROGRAMMING I

Chapter 10

2017

Topics

10.1 Pointers and the Address Operator

10.2 Pointer Variables

10.3 The Relationship Between Arrays and Pointers

10.4 Pointer Arithmetic

10.5 Initializing Pointers

10.6 Comparing Pointers

10.7 Pointers as Function Parameters

10.9 Dynamic Memory Allocation

10.10 Returning Pointers from Functions

Memory Address and Address Operator

- Each variable in a program is stored in a unique location of main memory.
  - Each location of the memory has "memory address".

- Use the address operator **&** to get the address of a variable:

```
int num = -23;
cout << &num; // prints address
              // in hexadecimal
```

- Note: & has three meanings in C++.
  - 1) reference variable if used next to data type in a declaration.
    - int &y = x;
  - 2) bitwise 'AND' operator if used in a binary expression
    - if (X & Y != 0) { }
  - 3) address operator

Reference & Memory Address

S/W

variable (reference): address

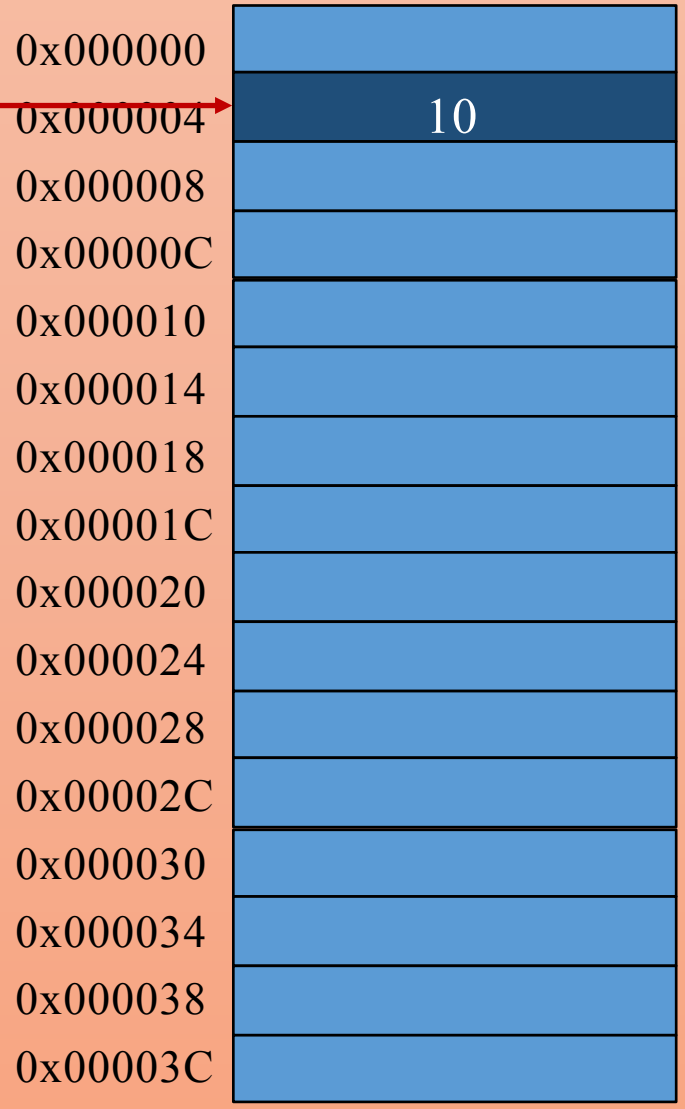
```
X(int)  :  0x00004
```

```
int X;

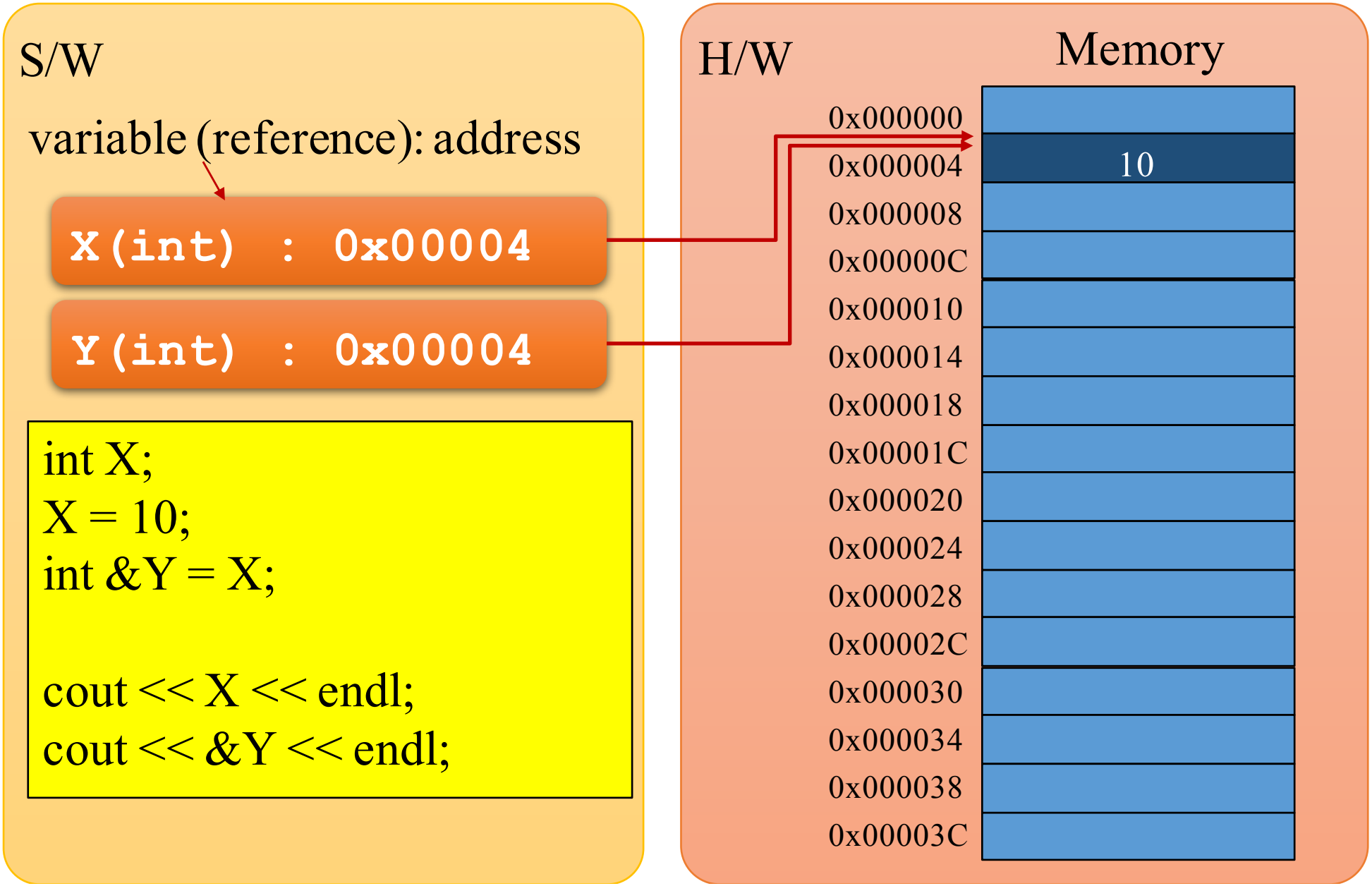
X = 10;

cout << X << endl;
cout << &X << endl;
```

H/W



Reference & Memory Address



Pointer Variables

- Pointer variable (pointer): variable that holds a memory address

- Definition:

```
int  *intptr;
```

- Read as:
  "`intptr` is a pointer that points to an int type variable"

- Spacing in definition does not matter:

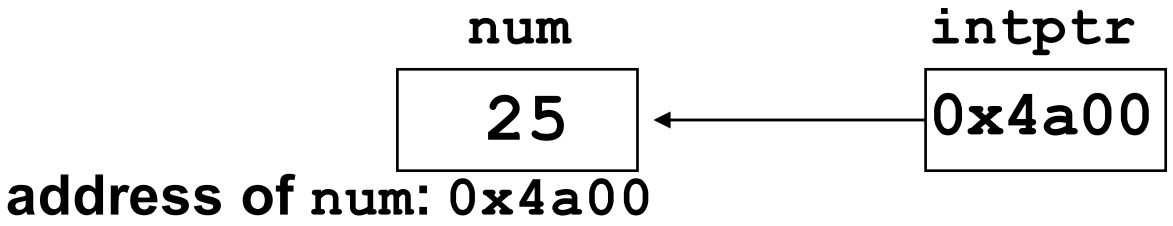
```
int * intptr;
int*  intptr;
```

Pointer Variables

- Assignment:

```
int num = 25;
int *intptr;
intptr = &num;
```

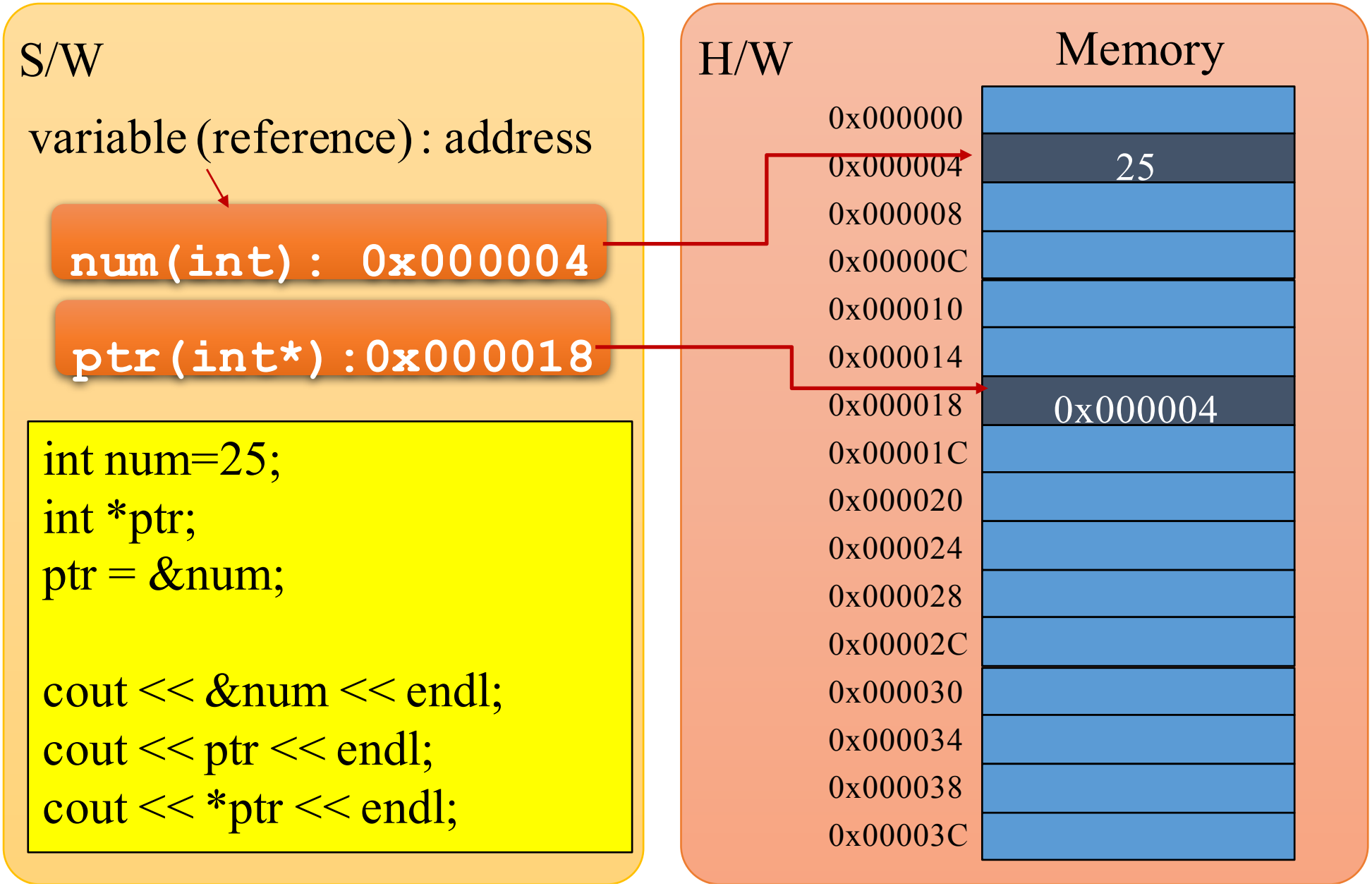
- Memory layout:



- Can access `num` using `intptr` and indirection operator `*`:

```
cout << intptr;  // prints 0x4a00
cout << *intptr; // prints 25
```

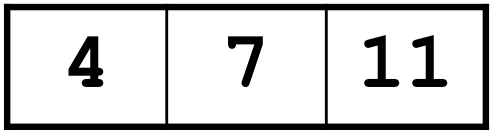
Pointer & Memory Address



Arrays and Pointers

- Array name is a variable that stores starting address of array

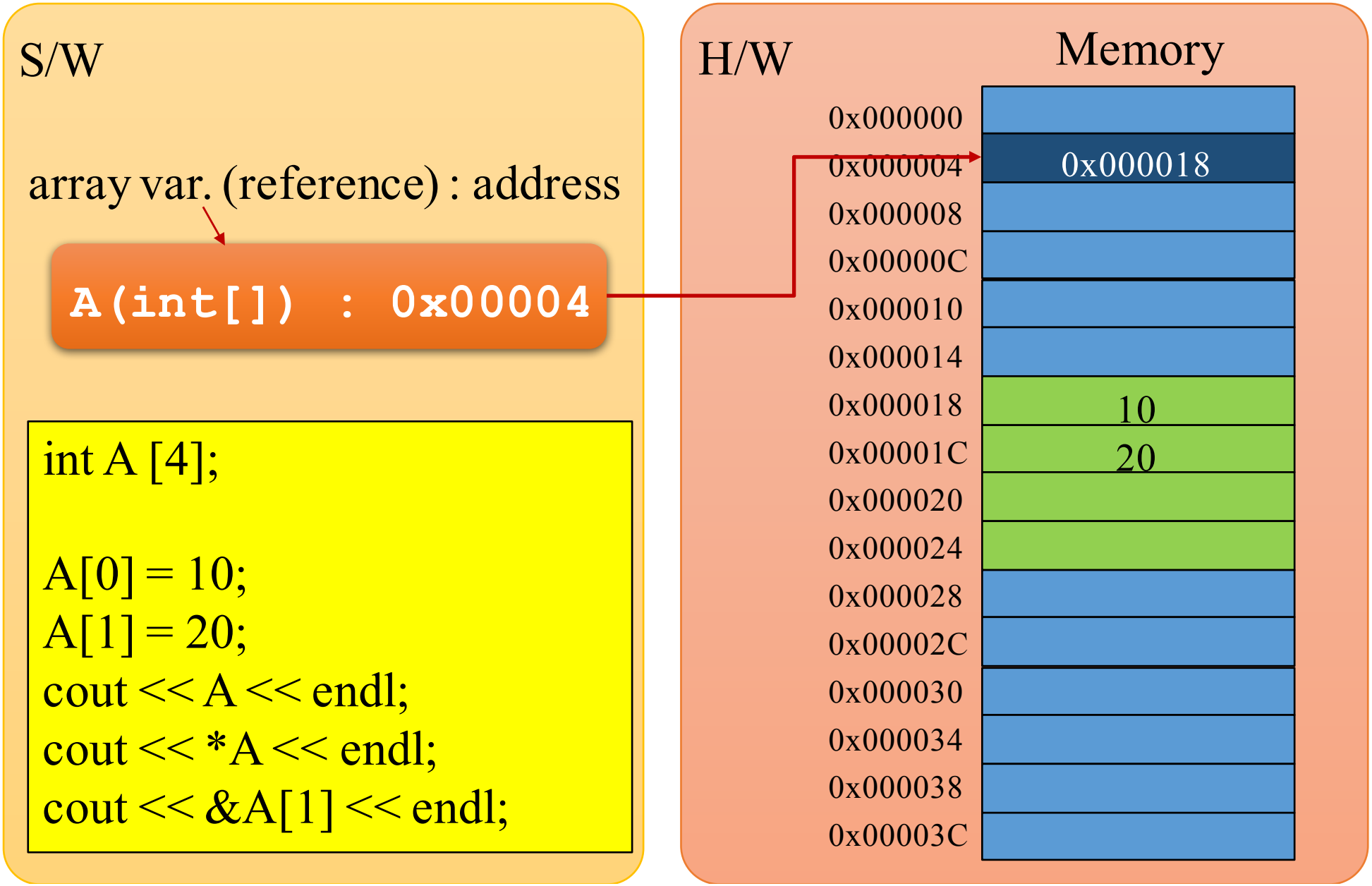
```
int vals[] = {4, 7, 11};
```



starting address of `vals`: `0x4a00`

```
cout << vals;    // displays 0x4a00
cout << vals[0]; // displays 4
```

Array & Memory Address



Arrays and Pointers

- Array name is a pointer constant

```
int vals[] = {4, 7, 11};
cout << *vals;    // displays 4
```

- Pointer can be used as an array name

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Pointers in Expressions

- Given:

```
int vals[]={4,7,11};
int *valptr = vals;
```

- `cout << (valptr + 1)` ?
- It means (address in `valptr`) + (1 * size of an `int`)

```
cout << *(valptr+1); // displays 7
cout << *(valptr+2); // displays 11
```

- Must use `( )` in expression

Array Access

- Array elements can be accessed in various ways

Array access method	Example
array name and `[ ]`	`vals[2] = 17;`
pointer to array and `[ ]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals+2) = 17;`
pointer to array and subscript arithmetic	`*(valptr+2) = 17;`

Array Access

- Array notation

```
vals[i]
```

  is equivalent to the pointer notation

```
*(vals + i)
```

- No bounds checking performed on array access

Pointer Arithmetic

Some arithmetic operators can be used with pointers:

- Increment and decrement operators ++, --
- Integers can be added to or subtracted from pointers using the operators +, -, +=, and -=
- One pointer can be subtracted from another by using the subtraction operator -

- ```
  int temp1[3];
  int temp2[3];
  cout << temp1 << endl;  // 0x00010
  cout << temp2 << endl;  // 0x0001C
  cout << (temp2 - temp1) << endl; // ?
  ```
  ```

Pointer Arithmetic

Assume the variable definitions

```
int vals[]={4,7,11};
int *valptr = vals;
```

Examples of use of ++ and --

```
valptr++; // points at 7
valptr--; // now points at 4
```

More on Pointer Arithmetic

- Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

- Example of the use of + to add an int to a pointer:

```
cout << *(valptr + 2)
```

This statement will print 11

More on Pointer Arithmetic

- Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

- Example of use of +=:

```
valptr = vals; // points at 4
valptr += 2;   // points at 11
```

Initializing Pointers

- Can initialize to NULL or 0 (zero)

```
int *ptr = NULL;
```

- Can initialize to addresses of other variables

```
int num, *numPtr = &num;
int val[ISIZE], *valptr = val;
```

- Initial value must have correct type

```
float cost;
int *ptr = &cost; // won't work
```

Comparing Pointers

- Relational operators can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2) // compares
                     // contents
```

Pointers as Function Parameters

- A pointer can be a parameter
- Works like a reference parameter to allow change to argument from within function
- A pointer parameter must be explicitly dereferenced to access the contents at that address

Pointers as Function Parameters

Requires:

1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr);
```

2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

3) address as argument to the function

```
getNum(&num);
```

Pointers as Function Parameters

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

int main(){
    int num1 = 2, num2 = -3;
    swap(&num1, &num2);
}
```

Dynamic Memory Allocation

- Q: Why do we need pointer variables?
- A: It allows us to allocate memory spaces while a program is running

- Uses **new** operator to allocate memory

```
double *dptr;
dptr = new double;
```

- **new** returns address of memory location

Dynamic Memory Allocation

- Can also use **new** to allocate array

```
arrayPtr = new double[25];
```

  - Program often terminates if there is not sufficient memory

- Can then use `[ ]` or pointer arithmetic to access array

Releasing Dynamic Memory

- Use **`delete`** to free dynamic memory

```
delete dptr;
```

- Use **`delete []`** to free dynamic array memory

```
delete [] arrayptr;
```

- Only use **`delete`** with dynamic memory!

Dangling Pointers and Memory Leaks

- A pointer is dangling if it contains the address of memory that has been freed by a call to **`delete`**.
  - Set such pointers to 0 as soon as memory is freed.

- A memory leak occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
  - Free up dynamic memory after use