a place of mind

THE UNIVERSITY OF BRITISH COLUMBIA

Pointers

C pointers and memory

Geoff's self-checklist:

- ~~Start iClicker Cloud~~
- Record lecture

Announcements

- No iClicker questions today!
- Lab1 Visual Studio issues
  - Project files for VS2019 provided
  - If you are working on other versions, you can create your own projects following instructions at https://www.students.cs.ubc.ca/~cs-259/vsprojecttutorial.html
  - For other issues, search/ask on Piazza
- Lab1 quiz next week
  - you must be registered in a lab section and attend your registered section (e-mail me otherwise)
  - open book/open notes (any printed materials)
  - Timed-release access according to lab section, and tight submission deadlines on GradeScope. More details to follow on Piazza
  - No resolution yet for DTS students

Function parameters

Pass by reference

- In some cases, parameters may be passed by reference ("call-by-reference")
  - The address (rather than the value) of the actual parameter is copied to the formal parameter when the function is called
  - Making a change to the value of the formal parameter effectively changes the value of the actual parameter
  - This is what occurs with array parameters, which are passed by reference by default

- More about this when we get to pointers

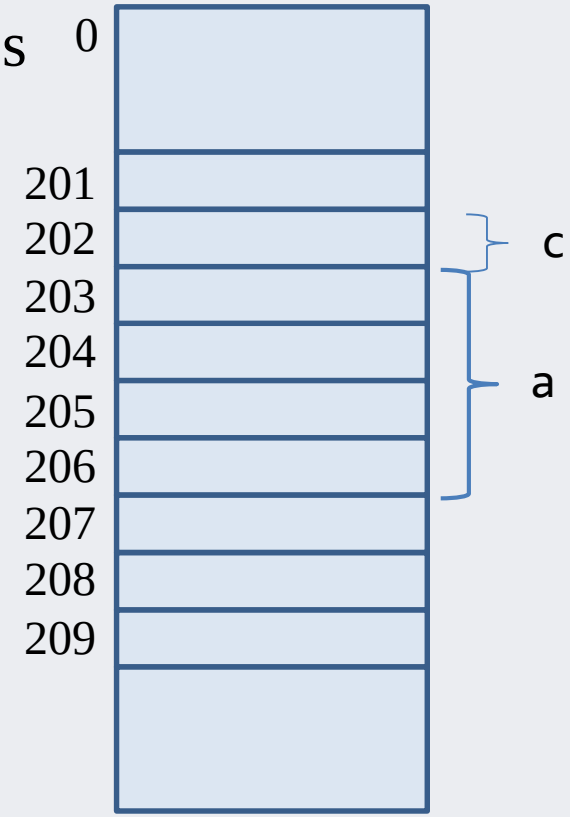
Addresses and pointers

- Every storage location in memory (RAM) has an *address* associated with it
  - The address is the location in memory where a given variable or identifier stores its data
- Can think of address in memory like a mailbox number
  - Use the address to find where the mailbox is
  - Look inside the mailbox to access the contents/value

Variable declaration

- Each byte of memory has a unique address

```
int a;
char c;
a = 5;
a++;
```



- At compile time, the compiler knows how much memory to allocate to each variable (e.g. 4 bytes for `int`, 1 byte for `char`, etc)

Addresses, &, and pointers

- You have already encountered addresses with the `scanf` function
  - `scanf` requires us to provide the address of a location using the "address of" operator, `&`
  - e.g. `scanf("%d", &a)`
  - This allows the `scanf` function to modify the value of the variable `a`, which is defined outside of `scanf`'s call stack

- A pointer is a data type that contains the address of the object in memory, but it is not the object itself

```
int a = 5;
int* p = &a;
```

  - `a` is an integer variable with the value 5
  - `p` is a pointer variable storing the address of `a`

Declaring pointers

- Pointer variables are declared as follows:

```
datatype* identifier
```

  - e.g. `int* ptr;` or `int * ptr;` or `int *ptr;`
- Note that the type of a pointer is not the same as the type it points to
  - e.g. `ptr` is a pointer to an `int`, but is itself not an `int`

- Warning! The declaration

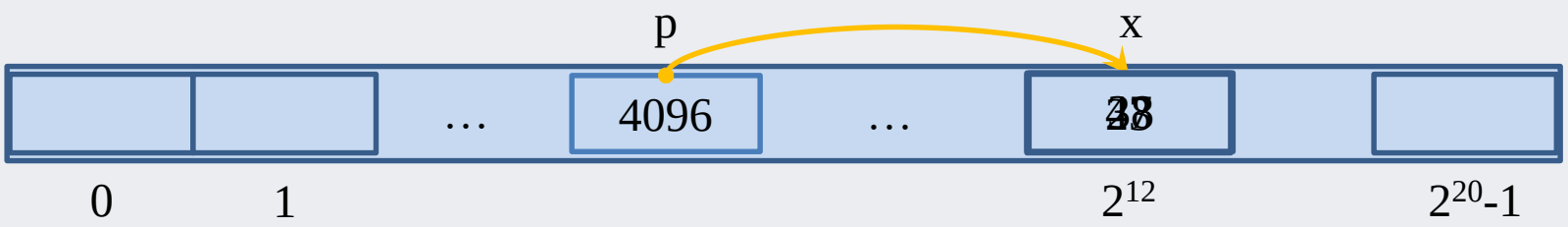
```
int* var1, var2;
```

  - declares `var1` as a pointer, but `var2` as an integer!
- To declare both as pointers, either declare individually, or:

```
int *var1, *var2;
```

Address operator and dereferencing

- Pointers can be assigned the address of an existing variable
  - Using the address operator, &
- The value which a pointer points to can be accessed by *dereferencing* the pointer
  - Using the * operator




```
int x = 23;
```

```
int* p = &x;
```

```
x = 47;
```

```
*p = 38;
```

Pointers as parameters

- Function parameters can be passed by reference using pointers

```
int getArraySum(int arr[], int size, int* pcount) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    if (arr[i] > 0) (*pcount)++;
    sum += arr[i];
  }
  return sum;
}
```

```
int numpositive = 0;
int numbers[] = {3, 7, -9, 5, -4};
int result = getArraySum(numbers, 5, &numpositive);
printf("Array sum: %d\n", result);
printf("Number of positive elements: %d\n", numpositive);
```

```
Array sum: 2
Number of positive elements: 3
```

Pointers as parameters

- What is output after the code on the right is executed? What is on the call stack for each function call?

```
void f1(int arg)
{
  arg = 22;
  printf("f1 arg: %d\n", arg);
}
```

```
void f2(int* arg)
{
  *arg = 410;
  printf("f2 arg: %d\n", arg);
}
```

```
int x = 45;

f1(x);
printf("x after f1: %d\n", x);

f2(&x);
printf("x after f2: %d\n", x);
```

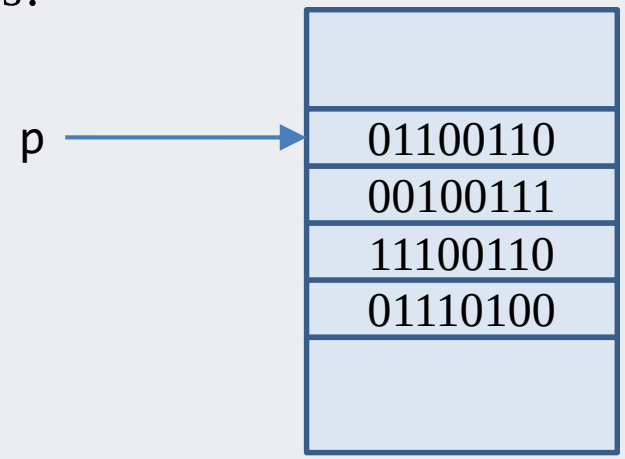
Modifying and dereferencing

- What is output by the following code?

```
int main() {
  int a = 5;
  int* p = &a; // assume 0x5fbff8cc

  printf("value of p = %p\n", p);
  printf("dereferenced p = %d\n", *p);

  p++;

  printf("value of p+1 = %p\n", p);
  printf("dereferenced p+1 = %d\n", *p);
}
```

Pointer types

- `int* p` is a pointer to an integer (at some 32-bit address)
- `char* p` is a pointer to a character (also at some 32-bit address)
  - Can we use a generic type for pointers?



- Can we dereference `p` without knowing the type of the variable that it is pointing to?
  - `int`: 4 bytes
  - `char`: 1 byte

Generic pointers

- Generic pointers can be declared, but must be cast before they can be dereferenced

```
int main() {
  int x = 10;
  char ch = 'A';
  void* gp;

  gp = &x;
  printf("integer value = %d\n", *(int*)gp); // outputs 10

  gp = &ch;
  printf("now points to char %c\n", *(char*)gp); // outputs A

  return 0;
}
```

Pointer to a pointer?

```
int main() {
  int x = 5;
  int* p = &x;
  *p = 6;
  int** q = &p;
  int*** r = &q;

  printf("%d\n", *p);
  printf("%d\n", *q);
  printf("%d\n", *(*q));
}
```

- "You can keep adding levels of pointers until your brain explodes or the compiler melts – whichever happens soonest"
  - stackoverflow user JeremyP

Back to call-by-reference

- Consider the following function that adds two parameters supplied by reference

```
int  add(int* num1, int* num2) {
  int sum = *num1 + *num2;
  return  sum;
}

int main() {
  int  a = 2;
  int  b = 4;
  int  c = add(&a, &b);
  printf("sum = %d",  c);
}
```

- Can we modify the add function so that it uses a pointer to return the answer?

Returning pointers

- Will it work if we just change the return type to pointer, and return the sum variable's address?

```
int* add(int* num1, int* num2) {
  int sum = *num1 + *num2;
  return &sum;
}

int main() {
  int  a = 2;
  int  b = 4;
  int* c = add(&a, &b);
  printf("sum = %d", *c);
}
```

This will have problems! Think about what is (or was) on the call stack

Passing array elements as parameters

- Arrays are passed by reference by default

```
double getMaximum(double data[], int size); // prototype
double getMaximum(double* data, int size); // equivalent prototype

double answer = getMaximum(myarr, length); // function call
```

- Note that we do not need to provide "&" when specifying the address of the entire array (i.e. the address of the first element)
- If we want to specify the address of an individual element of the array, we would need the address operator
  - e.g. `&data[4]`

Pointer arithmetic

- If we know the address of the first element of an array, we can compute the addresses of the other array elements
  - (or whatever comes after, if it is meaningful)

```
int A[5];
int* q = &A[0];
printf("q address: %p\n", q);
A[0] = 2;
A[1] = 4;

printf("value of q: %d\n", *q);
printf("value of q+1: %d\n", *(q + 1));
```

```
int x = 5;
int* p = &x;
printf("p address: %p\n", p);
printf("value of p: %d\n", *p);
printf("value of p+1: %d\n", *(p + 1));
```

Readings for this lesson

- Thareja
  - Chapter 1.11

- Next class – Dynamic memory allocation
  - Thareja – Appendix A