Slightly More Than Enough C++

44-440/640-IoT

Objectives

- Students will be able to
  - declare and dereference pointers
  - create arrays dynamically
  - work with C++ objects

Pointers

- def'n: in C++, a **pointer** is a variable that contains a memory address

Declaring a Pointer

- Pointers are declared by writing a * after the type

```
type * identifier;
```

- e.g.,

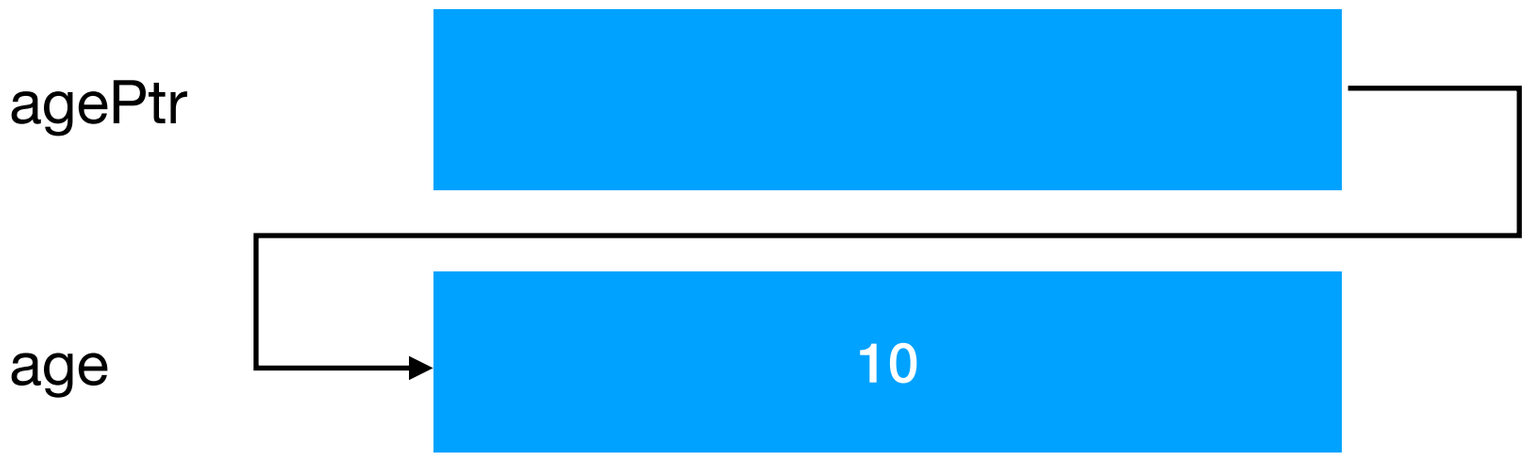
```
int * iptr;
float * fptr;
// That was easy :-)
```

- We will always read the * as "pointer", as in "int pointer iptr" and "float pointer fptr"

Assigning a Pointer a Value

- Pointers must be assigned the *address* of a variable.
- To get the address of a variable, precede it by an &, e.g., &age
- We will read & as "address of", as in "address of age"

```
int age = 10;
int * agePtr = &age;
```

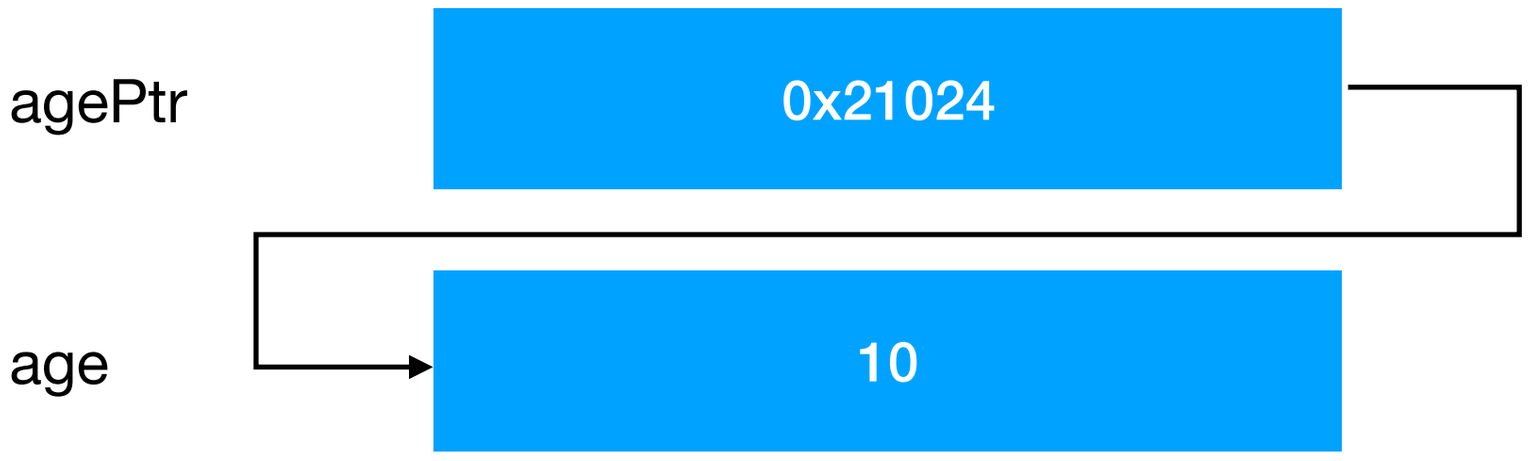


Pointers in Memory

- A pointer is a variable (4 bytes) and is stored in memory like any other variable.

- The only difference is that it contains an address

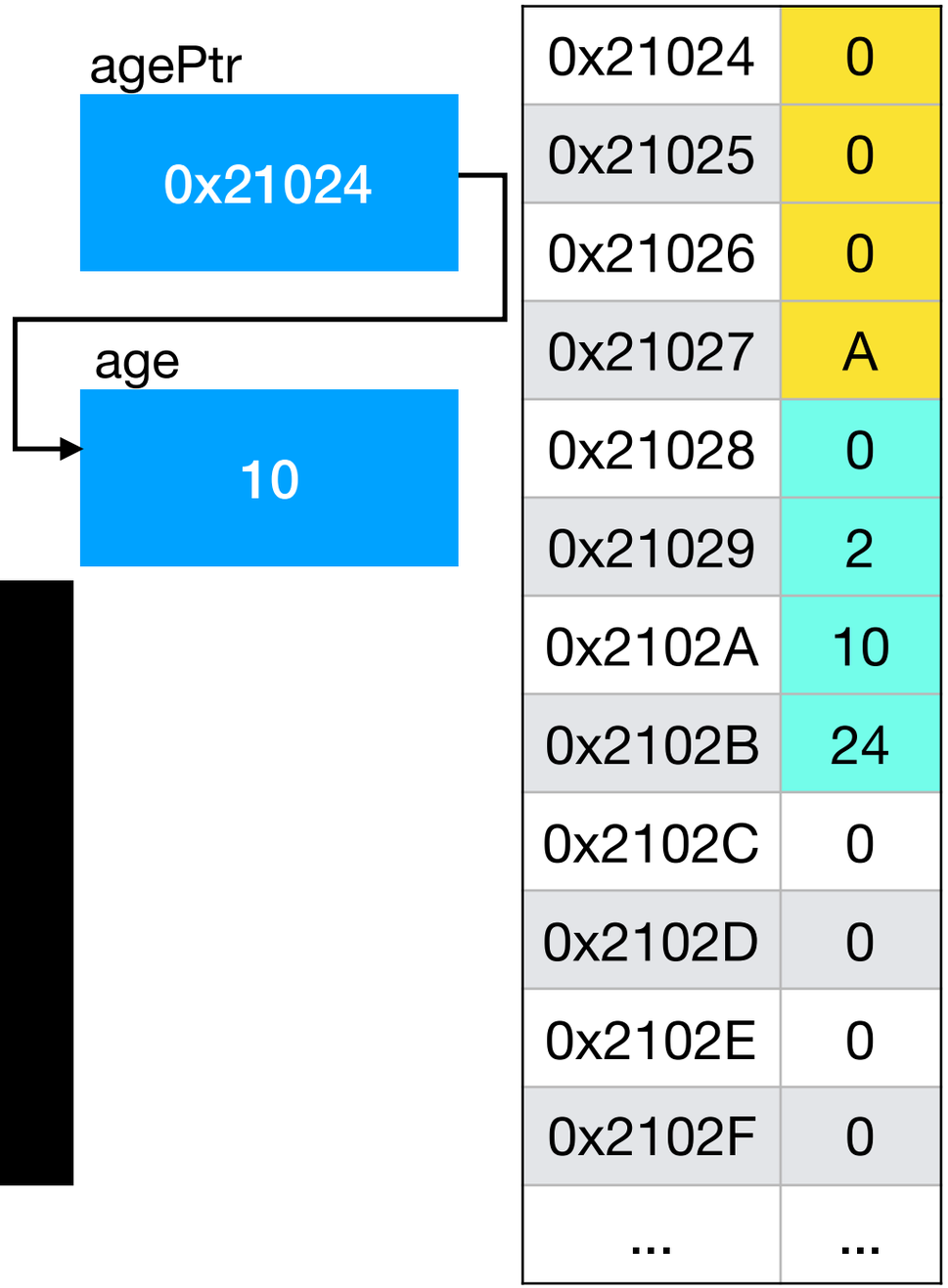
```
int age = 10;
int * agePtr = &age;
```



0x21024	0
0x21025	0
0x21026	0
0x21027	A
0x21028	0
0x21029	2
0x2102A	10
0x2102B	24
0x2102C	0
0x2102D	0
0x2102E	0
0x2102F	0
...	...

Dereferencing a Pointer

- To **dereference** a pointer, precede it with a *
- Dereferencing a pointer gets back to the memory location (the original variable)
- We will read the *, in this case, as "what's at", e.g., "what's at agePtr"




```
int age = 10;
int * agePtr = &age;

*agePtr = 20;
cout << age << ", " << *agePtr);

// output: 20, 20
```

Dereferencing a Pointer

- Dereference a pointer, to get back to the memory location, by preceding the pointer with a *. Placed there, read * as "what's at".

- e.g., if agePtr is a pointer to age, *agePtr **is** age, and you can do anything with *agePtr as you could do with age

```
int main(){
  int age;
  int * agePtr = &age; // age and *agePtr are now synonymous

  age = 11;
  *agePtr = 22; // just changed age to 22
  cout << agePtr << " " << *agePtr << " " << age;

  *agePtr = 33;
  *agePtr *= 2; // now it's 66 -- and there are a lot of *'s here...
  cout << agePtr << " " << *agePtr << " " << age;
   return 0;
}
```

```
output:
0x7ffeefbff538 22 22
0x7ffeefbff538 66 66
```

Predict the Output

```
int main(){

  float dist = 3.50f;
  float * distPtr = &dist;

  cout << "a: " << dist << endl;
  cout << "b: " << distPtr << endl;
  cout << "c: " << *distPtr << endl;

  *distPtr *= 2;

  cout << "d: " << dist << endl;
  cout << "e: " << distPtr << endl;
  cout << "f: " << *distPtr << endl;

  return 0;
}
```

Predict the Output

```
int main(){

  float dist = 3.50f;
  float * distPtr = &dist;

  cout << "a: " << dist << endl;
  cout << "b: " << distPtr << endl;
  cout << "c: " << *distPtr << endl;

  *distPtr *= 2;

  cout << "d: " << dist << endl;
  cout << "e: " << distPtr << endl;
  cout << "f: " << *distPtr << endl;

  return 0;
}
```

```
a: 3.5
b: 7edb1200
c: 3.5

d: 7
e: 7edb1200
f: 7
```

Icicle (a pointy ICE)

1. Declare a variable **flagHeight**, a double, with a value of 1.0.

2. Declare a pointer to flagHeight, **flagHeightPtr**, and assign it the address of flagHeight

3. Print out the value of flagHeight, using flagHeight

4. Print out the value of flagHeight, by dereferencing flagHeightPtr

5. Print out the value of flagHeightPtr (use %x)

6. Add 1.0 to flagHeight (using flagHeight)

7. Add 2.0 to flagHeight (by dereferencing flagHeightPtr)

8. Print flagHeight and the dereferenced flagHeightPtr out

9. Add 3.0 to flagHeightPtr and print out flagHeight by dereferencing flagHeightPtr. What happened?

10. Repeat step 9, this time using 3 (an int)

Icicle (a pointy ICE)

```
int main(){
        double flagHeight = 10.0;
        double * flagHeightPtr = &flagHeight;

        cout << flagHeight << " "
             << *flagHeightPtr << " "
             << flagHeightPtr << endl;

        flagHeight += 1.0;
        *flagHeightPtr += 2.0;

        cout << flagHeight << " "
             << *flagHeightPtr << endl;

        flagHeightPtr += 3;
        cout << *flagHeightPtr << endl;

        return 0;
}
```

Icicle (a pointy ICE) Solution

```
int main(){
        double flagHeight = 10.0;
        double * flagHeightPtr = &flagHeight;

        cout << flagHeight << " "
             << *flagHeightPtr << " "
             << flagHeightPtr << endl;

        flagHeight += 1.0;
        *flagHeightPtr += 2.0;

        cout << flagHeight << " "
             << *flagHeightPtr << endl;

        flagHeightPtr += 3;
        cout << *flagHeightPtr << endl;

        return 0;
}
```

```
10 10 0x7ffeefbff520
13 13
6.95329e-310
```

Dynamically Allocated Arrays

```
int main(){

    int numDataPoints;
    cout << "# of data points: ";
    cin >> numDataPoints;
    int * data = new int[numDataPoints];

    for(int i=0; i < numDataPoints; i++){
        cout << "Data Point # " << i << ": ";
        cin >> data[i];
    }

    for(int i=0; i < numDataPoints; i++){
        cout <<  data[i] << endl;
    }

    return 0;

}
```

new allocates room for an array of ints, and returns the address ... which, what luck, is just what a pointer is designed to store.

Predict the Output

```
void reset(int x);

int main(){
  int x = 10;
  reset(x);
  cout << x << endl;

 return 0;
}

void reset(int x){
  x = 0;
}
```

Predict the Output

```
void reset(int x);

int main(){
  int x = 10;
  reset(x);
  cout << x << endl;

  return 0;
}

void reset(int x){
  x = 0;
}
```

10

Why Use Pointers?

- To facilitate pass-by-reference, using functions, so we can return multiple values

- Efficiency when dealing with arrays and strings

Pass-by-value v. Pass-by-reference

- Pointers allow us to implement a pass-by-reference strategy.

- Normally, when you pass an argument into a parameter, it is **passed by value**: a separate memory space is allocated for that parameter, so changing the parameter has no effect on the argument

- Pass-by-reference means that we pass the address of an argument, and therefore can access that argument directly

Pass-by-reference: C

```
void reset(int x);

int main(){
  int x = 10;
  reset(&x);
  cout << x << endl;

 return 0;
}

void reset(int *x){
  *x = 0;
}
```

```
Now x is 0
```

Pass-by-reference: C++

```
void reset(int & x);

int main(){
  int x = 10;
  reset(x);
  cout << x << endl;

 return 0;
}

void reset(int & x){
  x = 0;
}
```

```
Now x is 0
```

Passing by reference happens so frequently that C++ includes a syntactical shortcut.

Here, the & means to pass the address of x into the parameter. You don't need to dereference it, though, that happens automatically

Other Helpful Libraries

- Containers
- Algorithms
- Strings
- Numerics

We will not cover these in detail in this class, but you need to study their APIs to make sure you can work with them. They may come in handy during some programming exercises.

Objects By Example

```
class Shape {
public:
    double x;
    double y;

    Shape(){}
    Shape(int x, int y){
        this->x = x;
        this->y=y;
    }
    virtual double area(){return 0.0;}
private:
    string copyright = "2018";
};                      // This ; is needed here
```

- Access: public, protected, private
- virtual means the method can be overridden
- Short methods can be defined inline. For longer methods, it is better to specify just the prototype and provide the implementation elsewhere

```
// Classes in C++ are vaguely like they are in Java.
// This is just to show the syntax & get you started
// thinking about it.
class Circle : Shape {
public:
    double x;
    double y;
    double radius;
    static const int numCircles = 0; // only with ints
    static constexpr double pi = 3.141592654;

    // The Circle constructor calls Shape's
    Circle(double x, double y, double radius) : Shape(x,y){
        this->radius = radius;
    }

     double area(){
         return pi * radius * radius;
    }

    void shrinkIt(double amount);

};

void Circle::shrinkIt(double amount){
    radius -= amount;
}
```

```
int main(int argc, const char * argv[]) {
    Circle circ = Circle(20, 50, 22);
    Circle * circ2 = &circ; // both circ & circ2 point to the same Circle

    (*circ2).radius = 25.0;
    circ2 -> radius = 25.0; // same, but more esthetically pleasing
     return 0;
}
```

Solution

Resources

- http://wiki.wiring.co/wiki/C/C%2B%2B_Comparison
- http://en.cppreference.com/w/
- http://en.cppreference.com/w/cpp/language/types
- https://community.particle.io/t/how-pure-is-the-c-in-photon/23438/4