Functions and Pointers

Variables

Java	C++

▪ All primitives are values
▪ All objects are references
▪ There are no pointers
	▪ all variables are either
	  ▪ values
	  ▪ int x;
	  ▪ references
	  ▪ float &y;
	  ▪ pointers
	  ▪ char* z;

Functions

Java	C++

▪ void primitive(int x) {
// the argument passed to
// primitive is copied to
// the stack variable x
}
▪ void object(Object y){
// y is a reference to
// the argument passed to
// to the method object
}
	▪ void anyFunction(type x) {
	// the argument passed to
	// anyFunction is copied to
	// the stack variable x
	}
	▪ void anyFunction(type &y){
	// y is a reference to
	// the argument passed to
	// to anyFunction
	}

Functions

Java	C++

▪ int primitive(int x) {
// the argument passed to
// primitive is copied to
// the stack variable x
}
▪ Object object(Object y){
// y is a reference to
// the argument passed to
// to the method object
}
	▪ int anyFunction(type x) {
	// the argument passed to
	// anyFunction is copied to
	// the stack variable x
	return 0;
	}
	▪ type anyFunction(type &y){
	// y is a reference to
	// the argument passed to
	// to anyFunction
	type x = y;
	return x; // note return a value
	}

The **NULL** Pointer

- C guarantees that zero is never a valid address for data
- A pointer that contains the address zero known as the **NULL** pointer
- It is often used as a signal for abnormal or terminal event
- It is also used as an initialization value for pointers

Pass by Value

- All functions are pass-by-value in C
  - A copy is made of each parameter's value and then the copy is passed
  - In C, variables supplied as parameters to a function call are protected against change
    - i.e. impossible to write a `swap(x, y)` function
  - Only way to modify a variable through a function is to assign the return value to that variable
- However C++ adds pass-by-reference!!!
  - when you pass by reference the reference parameters are not protected against change
  - i.e. possible to write a `swap(x, y)` function

Pass by Value and Pointers

- Some functions are pass-by-value in C++
- Pass-by-value still holds even if the parameter is a pointer
  - A copy of the pointer's value is made – the address stored in the pointer variable
  - The copy is then a pointer pointing to the same object as the original parameter
  - Thus modifications via de-referencing the copy STAYS.

Pass by Reference and Pointers

- Other functions are pass-by-reference in C++
- Pass-by-reference still holds even if the parameter is a pointer
  - A reference of the pointer is passed– the address stored in the pointer can change
  - The reference is then a pointer pointing to the same object as the original parameter
  - Thus modifications via de-referencing the copy STAYS.
  - In addition, you can modify the original pointer.

Function Arguments

- **x** and **y** are copies of the original, and thus **a** and **b** can not be altered.

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

int main() {
  int a = 1, b = 2;

  swap(a, b);
  return 0;
}
```

Wrong!

Function Arguments

- **x** and **y** are references to the original, and thus **a** and **b** can be altered.

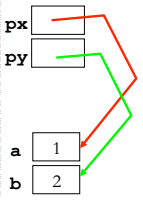
```
void swap(int &x, int &y) {
  int tmp;
  tmp = x; x = y; y = tmp;
}

int main() {
  int a = 1, b = 2;

  swap(a, b);
  return 0;
}
```

Right!

Pointers as Function Arguments

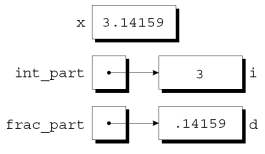
- Passing pointers – **a** and **b** are passed by reference (the pointers themselves **px** and **py** are still passed by value)

```
void swap(int *px, int *py) {
  int tmp;
  tmp = *px; *px = *py; *py = tmp;
}

int main() {
  int a = 1, b = 2;

  swap(&a, &b);
  return 0;
}
```



Pointers as Function Arguments

- Write a function that will decompose a `double` value into an integer part and a fractional part.
- As a result of the call, `int_part` points to `i` and `frac_part` points to d:



Pointers as Function Arguments

```
void decompose(double d, int *i, double *frac) {
  *i = (int) d;
  *frac = d - *i;
}

int main() {
  int int_part;
  double frac_part, input;

  cin >> input;
  decompose(input, &int_part, &frac_part);
  cout << input << "decomposes to " <<
    *int_part << " and " << *frac_part << endl;
  return 0;
}
```

Pass by Reference

- In C:
  - The pointer variables themselves are still passed by value
  - In a function, if a pointer argument is de-referenced, then the modification indirectly through the pointer will stay
- In C++:
  - using the reference type (&) allows the copying of pointers and de-referencing to be invisible to the user. (Syntactic sugar)

Pointers are Passed by Value

```
void f(int *px, int *py) {
  px = py;
}

int main() {
  int x = 1, y = 2, *px;
  px = &x;
  f(px, &y);
  cout << *px << endl;
}
```

Modification of a Pointer (in C)

```
void g(int **ppx, int *py) {
  *ppx = py;
}

int main() {
  int x = 1, y = 2, *px;
  px = &x;
  g(&px, &y);
  cout << *px << endl;
}
```

Modification of a Pointer (in C++)

```
void g(int *&ppx, int *py) {
  ppx = py;
}

int main() {
  int x = 1, y = 2, *px;
  px = &x;
  g(px, &y);
  cout << *px << endl;
}
```

Pointer as Return Value

- We can also write functions that return a pointer
- Thus, the function is returning the memory address of where the value is stored instead of the value itself
- Be very careful not to return an address to a temporary (stack) variable in a function!!!

Example

- **x** and **y** are copies of the original, and thus what is **&x** and **&y**?

```
int* max(int *x, int *y) {
  if (*x > *y) {
    return x;
  }
  return y;
}

int main() {
  int a = 1, b = 2, *p;

  p = max(&a, &b);
  return 0;
}
```

```
int* max(int x, int y) {
  if (x > y){
    return &x;
  }
  return &y;
}

p = max(a, b);
```

Reference as Return Value

- We can also write functions that return a reference
- Thus, the function is returning a reference to the value.
- Be very careful not to return a reference to a temporary (stack) variable in a function!!!
- You should only ever return a reference that was passed into the function, or a reference to a global variable.
- If you return a reference to memory on the heap, then you need to make sure that the reference loses scope before the memory is deleted.

Example

- **x** and **y** are references of the original, and thus what is **returned**?

```
int& max(int &x, int &y) {
  if (x > y) {
    return x;
  }
  return y;
}

int main() {
  int a = 1, b = 2, *p;

  p = max(a, b);
  return 0;
}
```

```
int& max(int x, int y) {
  if (x > y){
    return x;
  }
  return y;
}

p = max(a, b);
```

Arrays as Arguments

- Arrays are passed by reference
- Modifications stay

```
/* equivalent pointer alternative */
void init(int *a) {
  int i;

  for(i = 0;i<SIZE;i++){
    *(a+i) = 0;
  }
}
```

```
#define SIZE 10

void init(int a[]) {
  int i;

  for(i = 0;i<SIZE;i++){
    a[i] = 0;
  }
}

int main() {
  int a[SIZE];

  init(a);
  return 0;
}
```

22

Array Arguments

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified:

```
int f(int a[]){  /* no length specified */
  …
}
```

- We can supply the length—if the function needs it—as an additional argument.

Array Arguments

- Example:

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

Array Arguments

- The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

- We can omit the parameter names if we wish:

```
int sum_array(int [], int);
```

Array Arguments

- When `sum_array` is called, the first argument will be the name of an array, and the second will be its length:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

- Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);   /*** WRONG ***/
```

Array Arguments

- Suppose that we've only stored 50 numbers in the `b` array, even though it can hold 100.
- We can sum just the first 50 elements by writing

```
total = sum_array(b, 50);
```

- Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150); /*** WRONG ***/
```

`sum_array` will go past the end of the array, causing undefined behavior.

Array Arguments

- A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument.
- A function that modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

Array Arguments

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.
- If we revise `sum_array` so that `a` is a two-dimensional array, we must specify the number of columns in `a`:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```