Lecture 6

C++ Programming

Arne Kutzner
Hanyang University / Seoul Korea

Function overloading

Function Overloading

- Two functions have the **same name**
- Functions **can be distinguished by types** of function header
- Example:

```
double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
int max(int num1, int num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

Same name but different types

Function Overloading

- During compile time we can select the appropriate definition of a overloaded function

```
d = max(2.0, 3.0);

i = max(2, 3);
```

```
double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

```
int max(int num1, int num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

Warning: Ambiguous Invocation

- Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match.
- This is referred to as ambiguous invocation.
- Ambiguous invocation is a compilation error, but can be resolved by a type-cast.

Ambiguous Invocation Example

```
double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
}
```

?

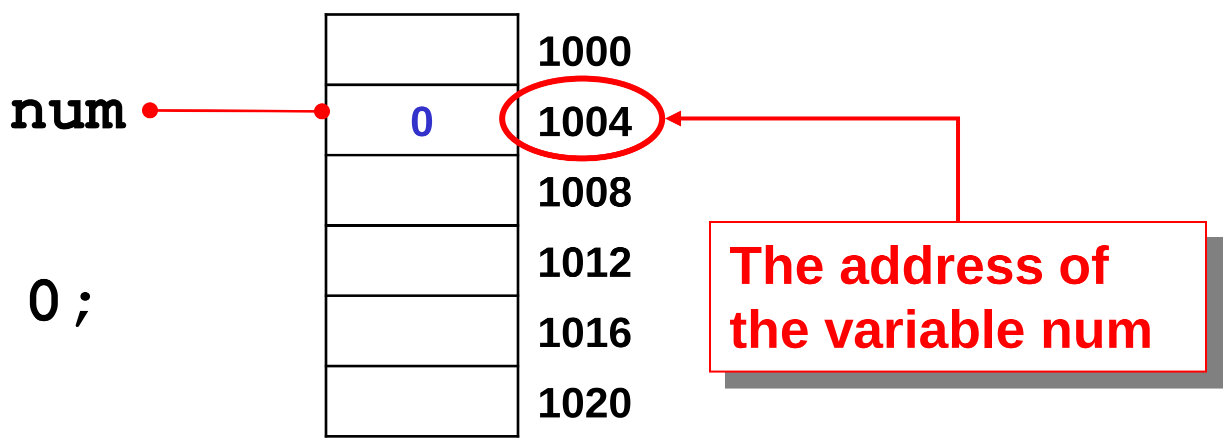
```
d = max(4.0, 5.0);
```

```
double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
}
```

Pointers / Introduction

Variables and Memory Address

- Every variable has some location in the memory that holds the value of the variable. This location is called the **address** of the variable.




```
int num = 0;
```

Pointers

- A **pointer** is also a variable, but unlike the other types of variables that we have seen, it **contains a memory address as its value**.

Pointer Declaration

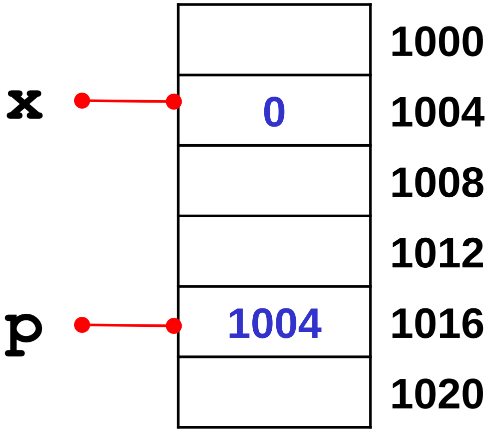
- To declare a pointer variable, a asterisk(**`*)`** should placed in front of name of the name of the variable. Example: **`int *p;`**

`p` is a pointer to an integer value (i.e. **`p`** contains the address of some memory cell that contains an integer value)

Address Operator

- You can get the memory address of some variable by using the **address operator `&`**. Example:

```
int x = 0;
int *p;
p = &x;
```



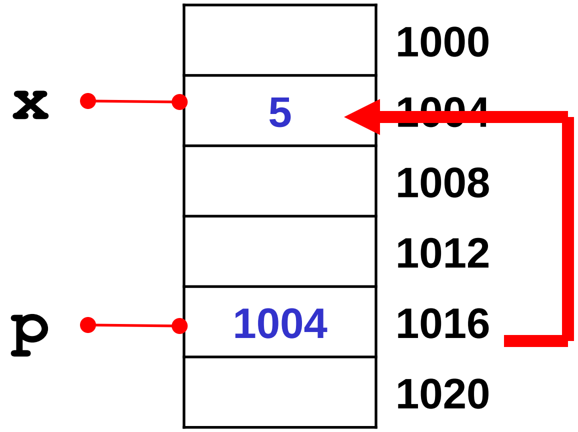
The pointer `p` gets the address of the variable `x` as its value

How to dereference pointers?

- For dereferencing a pointer place a asterisk in front of the pointer variable name. Example:

```
int x = 0;
int *p;
p = &x;
*p = 5;
```

Write the value 5 to the memory location referenced by **p**



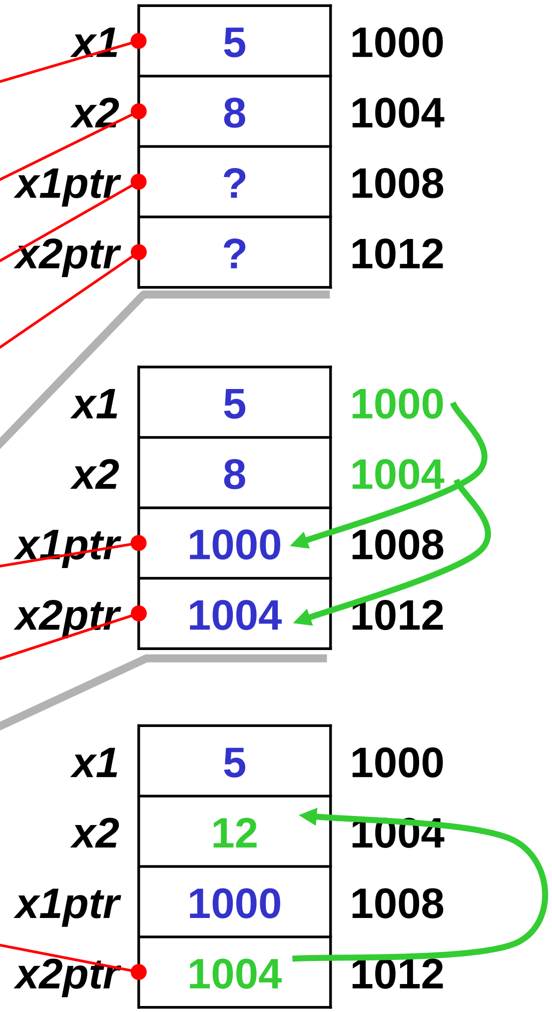
Pointer Types

- Pointers can be declared to point to variables of any type.
- Examples:

`int *p1;`	pointer to an integer
`double *p2;`	pointer to a floating point number
`char *p3;`	pointer to a character

Example Program 1

```
int main() {
...
int x1 = 5;

int x2 = 8;

int *x1ptr;

int *x2ptr;

x1ptr = &x1;

x2ptr = &x2;

*x2ptr = 12;
...
}
```



Example Program 2

```
int main() {
int num1, num2, product;
int *ptr_num1, *ptr_num2;

num1 = 5;
num2 = 7;

ptr_num1 = &num1;
ptr_num2 = &num2;

product = num1 * (*ptr_num2);
 …
 }
```

The resulting memory picture

num1	**5**	**1000**
num2	**7**	**1004**
product	**35**	**1008**
ptr_num1	**1000**	**1012**
ptr_num2	**1004**	**1014**

***** operator specifies **dereferencing a pointer**

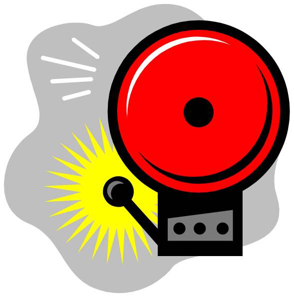
***** operator specifies **multiplication**

Pointer type equivalence

- A pointer can dereference only a variable which is of the same type. Example:

```
int n = 5;
double x = 5.0;
int *ptr1;
double *ptr2;
ptr1 = &n;
ptr2 = &n;
ptr1 = &x;
```

Compile error!
ptr1 is a pointer to integer whereas x is a floating point number.
ptr2 is a pointer to double whereas n is an integer number.



Warning: Uninitialized Pointers

- Dereferencing a pointer that has not been properly initialized

  Example:

The pointer contains an arbitrary address, so **`*p=1`** may overwrite some arbitrary location in memory which can cause severe problems.

```
int *p;

*p = 1;
printf("Value is %d", *p);
```

Effects with uninitialized/wrong pointers

- Program attempts to access some random memory location.
  Possible effects:
  - Fatal Runtime error (common program run-time error message: "segmentation fault").
  - Accidentally modifying other data / variables. Program continues but delivers incorrect result.

Functions and Pointers

Pointers and Functions

- Function arguments can be of some “pointer type”. Example:

```
void main () {
   int i;

   fun (&i);
   std::cout << i;
}

void fun (int *p) {
   *p = 1;
}
```

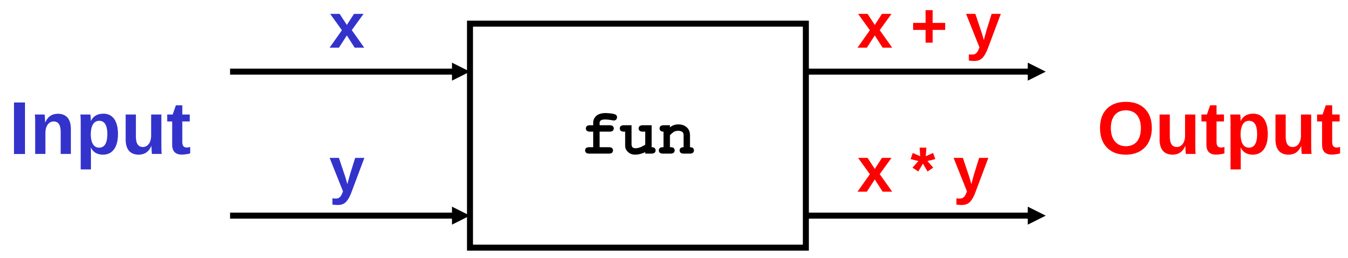
Type of function arguments is “pointer to int”

Call by Reference

- In the example on the slide before the statement `*p = 1` in `fun` changes the value of `i` defined `main` in.
- Because `p` *refers* `i` in `fun` we call this form of argument passing **call by reference**.

Example for the Application of Call by Reference

- A function **`fun`** shall have two input arguments and two output argument.



- **For call by reference we can use in C++ <u>reference parameter</u> as well as pointers**

Application of Call by Reference

```
int main() {
   int x, y, sum, product;
   x = 3;
   y = 5;
   fun (x, y, &sum, &product)
}

void fun (int a, int b, int *p1, int *p2)
{
   *p1 = a + b;
   *p2 = a * b;
}
```

The addresses of sum and product are passed as arguments.

Application of Call by Reference

- In the previous example, the addresses (not values) of the variables `sum` and `product` are copied to the function pointer arguments `p1` and `p2`. In this way, `sum` and `product` have the same memory locations as `*p1` and `*p2`, respectively.

x	3	
y	5	
p1	1476	
product	15	1468
p2	1468	
sum	8	1476
	….	
a	3	
b	5	

Comparison of Call by Reference and Call by Value

- With the function call by value, the values of the function arguments can't be modified by the function.
- With the function call by reference, the function arguments are the addresses, not the values of the corresponding variables. These addresses are copied into some pointers which are function arguments. Therefore, modifying the values stored in the addresses pointed by these pointers modifies also the values of the variables.

Comparison of Call by Reference and Call by Value

What outputs do we get and why?

Call by value	***Call by reference***

…
`int n = 1;`
`int m = 2;`
 	
`swap(n, m);`
`Std::cout << n << " " << m;`
…
 	
`void swap(int x, int y)`
`  int temp = x;`
`  x = y;`
`  y = temp;`
`}`
	…
	`int n = 1;`
	`int m = 2;`
	
	`swap(&n, &m);`
	`std::cout << n << " " << m;`
	…
	
	`void swap(int *p1, int *p2)`
	`  int temp = *p1;`
	`  *p1 = *p2;`
	`  *p2 = temp;`
	`}`

Pointers and Arrays / Pointer Arithmetic

Pointers and Arrays

- In C there is a strong relationship between the concepts of pointers and arrays
- An array name is basically a *const* pointer. (A pointer with fixed address)

```
int *x;
int a[10];
x = a;
```

This statement is OK!!!
It assigns **`x`** the address of the first elements array **`a`**.

Pointers and Arrays (cont.)

- You can even use the [ ] operator with a pointer:

```
int *x;
int a[10];
x = a;
std::cout << x[2];
```

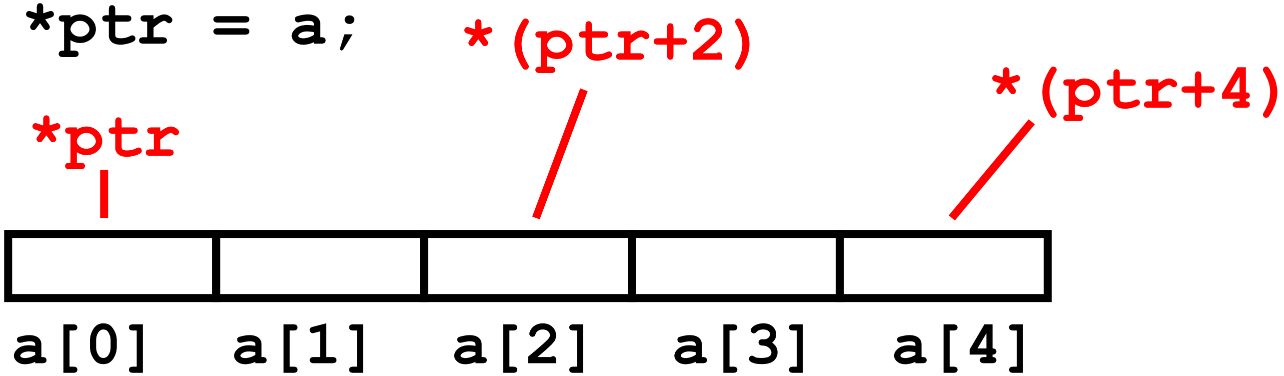
x gets "the address of the first element of **a**"

is identical to **a[2]**

Pointer arithmetic

- Integer math operations can be used with pointers.
- If you increment a pointer, it will be increased by the size of whatever it points to.

```
int *ptr = a;
```




```
int a[5];
```

printing an array

```
void print_array(int a[], int len) {
  for (int i = 0;i < len; i++)
     std::cout << i << " " << a[i];
}
```

array version

```
void print_array(int *a, int len) {
  for (int i = 0;i < len; i++)
   std::cout << i << " " << *a++;
}
```

pointer version

Arrays of Pointers

- Like for any other type you can create arrays of pointers:

```
int a[] = {0, 1, 2};
int b[] = {4, 5, 6};
int* x[2];
x[0] = a;
x[1] = b;

std::cout << *(x[1]+1);
Std::cout << *(*(x+1)+1);
```

x is array of pointers

both statements are identical

Matrices and Pointers …

- The technique can be extended to multidimensional arrays:

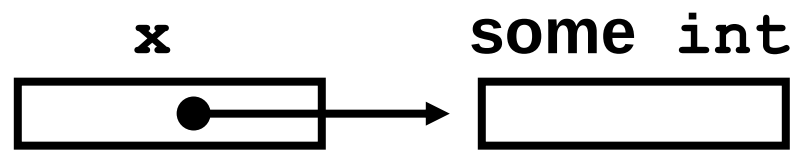
```
int a[2][3] = {{0, 1, 2},
               {4, 5, 6}};
int* x[2];
x[0] = a[0];
x[1] = a[1];

std::cout << *(x[1]+1);
std::cout << *(*(x+1)+1);
std::cout << a[1][1];
std::cout << *(a[1]+1);
std::cout << *(*(a+1)+1);
```

all statements deliver the same element

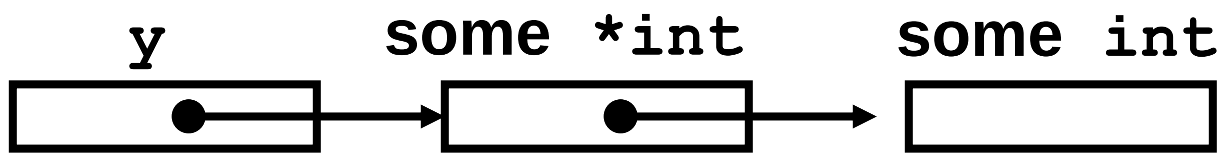
Pointers to Pointers

- Pointer can refer pointers …

```
int *x;
```



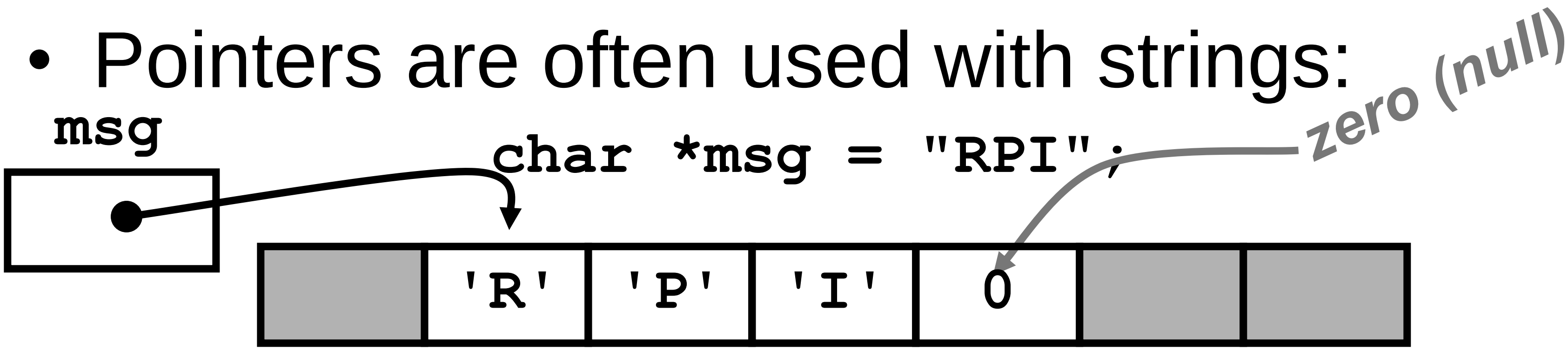

```
int **y;
```



C strings in C++

- A *string* is a *null terminated* array of characters.
  - null terminated means there is a character at the end of the the array that has the value 0 (null).
- Pointers are often used with strings:

```
char *msg = "RPI";
```



String Example - Count the chars

```
int count_string( char *s) {
  int n=0;
  while (*s) {
     n++;
     s++;
  }
  return(n);
}
```

while the thing pointed to by **s** is not null

increment count

set **s** to point to the next char

Another way

```
int count_string(char *s) {
  char *ptr = s;
  while (*ptr) {
    ptr++;
  }
  return(ptr - s);
}
```

pointer arithmetic!

The Heap /
Dynamic Memory Allocation /

`sizeof(...)` function

- Using the **`sizeof`** function we can get the number of bytes consumed by instances of some datatype
  Syntax: **`sizeof(`**<*datatype name*>**`)`** *;*
  Example:

```
std::cout << sizeof(int);
```

The Heap

- The **Heap** is a special area of memory that allows the **dynamic allocation of memory during Runtime**.
- Possible Application:
  Arrays with dynamic size, e.g. arrays where we know the size only during runtime (Remember: So far we have to specify the size (number of elements) of some array in the context of its declaration. E.g.: **`x[10]`**)

Heap Management in C

- **<u>We will later learn a second form of heap management in C++</u>**
- Old C-style heap management:
  Three functions for heap management defined in **`stdlib.h`**
  - **`calloc(n, el_size)`** – allocation of continuous space for an array of **`n`** elements with **`el_size`** bytes for each array element
  - **`malloc(size)`** – allocation of an area of **`size`** bytes in the heap
  - **`free(p)`** – freeing of allocated heap-space. (C stores internally the amount of memory associated with **`p`**)

"void" pointers in C and C++

- The following code triggers an error message in C++ but not in C

```
void* fun () {
   return NULL;
}
void main() {
   int *p;
   p = fun();
}
```