Murdoch
UNIVERSITY

Data Structures and Abstractions

Pointers & Parameters

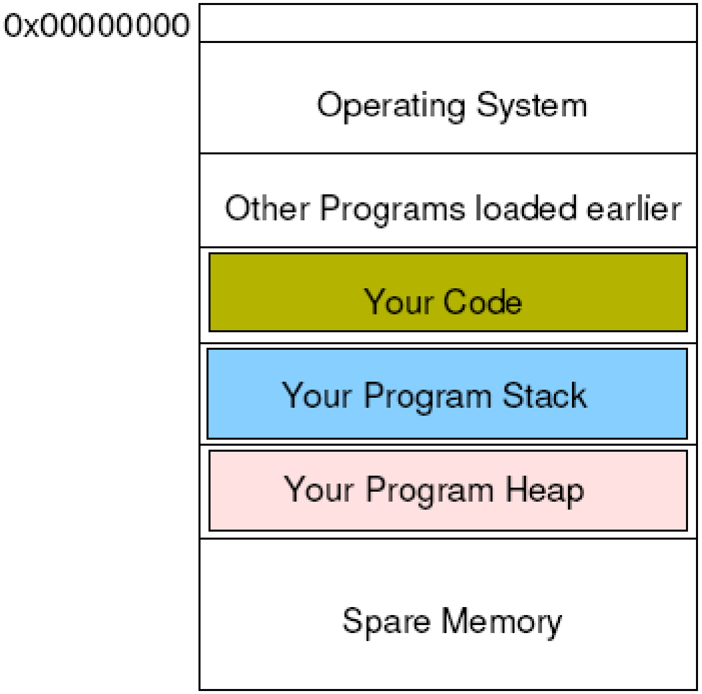
Lecture 2

Error Types Reminder

- Don’t forget the different error types when coding.
- Debugging is an absolute necessity when coding and even more so when coding pointers, so discussing this briefly here is appropriate.
- There are five types of common errors:
  - *Syntax* errors are those that prevent your code from compiling because of incorrect use of language grammar.
  - Semantic errors are errors in meaning. For example, if you have an apple object and you try to add to an orange object. Much more common ones are when you try to assign a float value to an integer variable. These can also be called type errors.
  - When your code compiles and your program runs as planned, but the output is incorrect, you have a *logic* error. That is why you need a test plan! And why you actually run through it!
  - When your program crashes it is a *run-time* error. You need a test plan and testing.
  - Finally if the input data is incorrect in some way, then your output will be incorrect also: a GIGO error.

RAM

- When your program runs, the OS allocates RAM for its use. [1]
- This RAM is divided up into different sections for use in different ways by your program. The layout shown below can be different for different architectures and OS. Covered in your first year unit.
- The program stack stores variables, parameters etc. that are defined when you write your program.
- The program heap is used for memory locations that are defined as your program runs: *dynamically* allocated variables.



0x00000000

Operating System

Other Programs loaded earlier

Your Code

Your Program Stack

Your Program Heap

Spare Memory

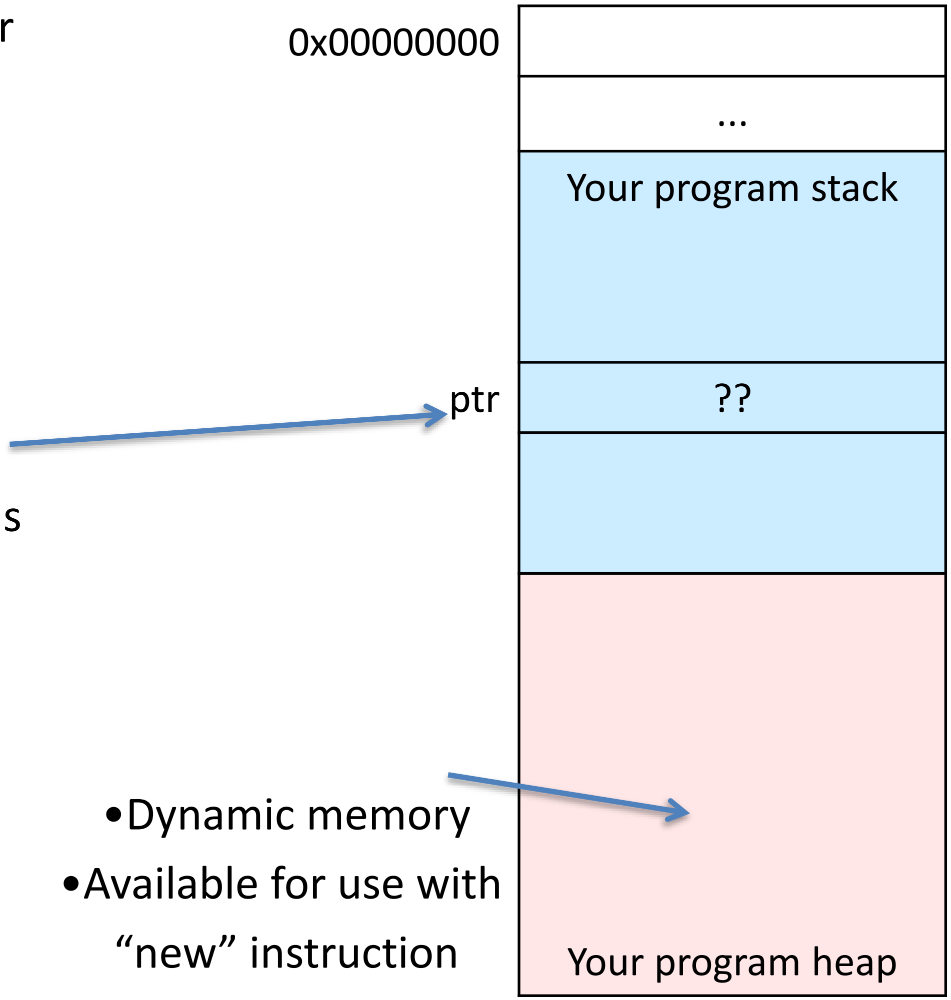
Pointers

- Pointers cause more heartache than is really necessary. [1]
- All they are is a variable that stores the address of another variable.
- Of course it is also possible to continue this ad-infinitum, which means that you can end up with an address, of an address, of an address...
- But in actually fact it is rare that you go beyond one, or two, levels of *dereference.*

Declaring Pointers

- This declares a pointer to an integer sized memory location.
- It does *not* allocate the memory location for you to store the actual data.
- Memory to store ptr itself but not the data. [1]
- Contents of ptr is whatever happens to be there by chance.

```
int f1()
{ // local vars on stack
        int *ptr;
}
```

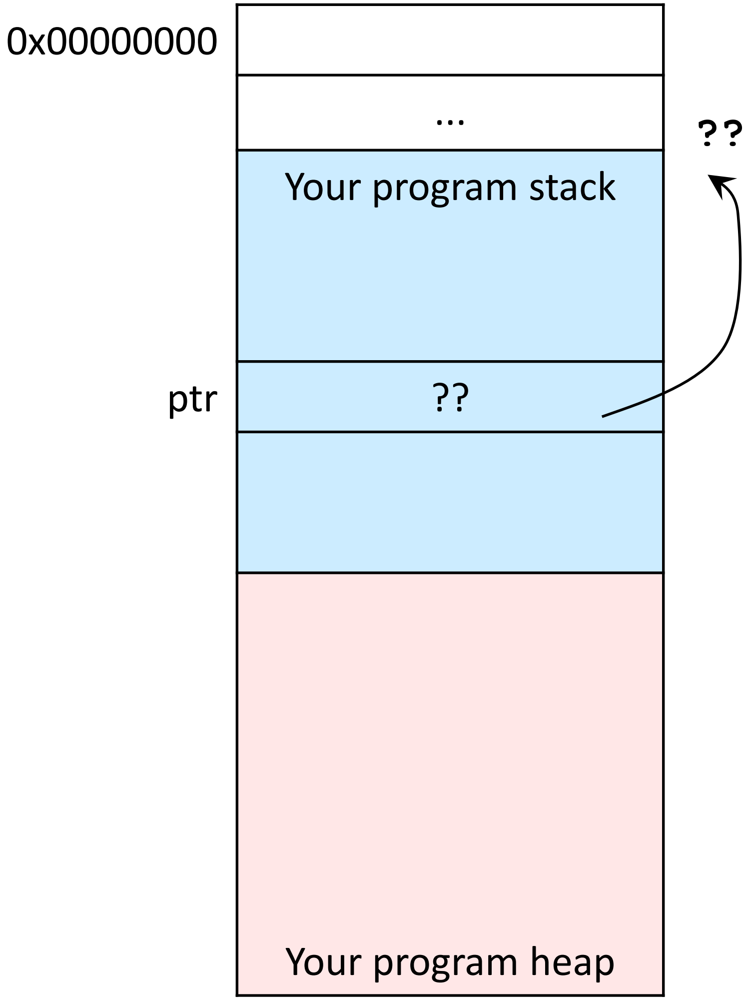


- Dynamic memory
- Available for use with “new” instruction

Declaring Pointers

- This means that trying to use the memory location contained in the pointer variable will crash the program or cause strange events.
- This is because the contents of ptr could be anything; i.e. it could be pointing *anywhere* in memory. [1]

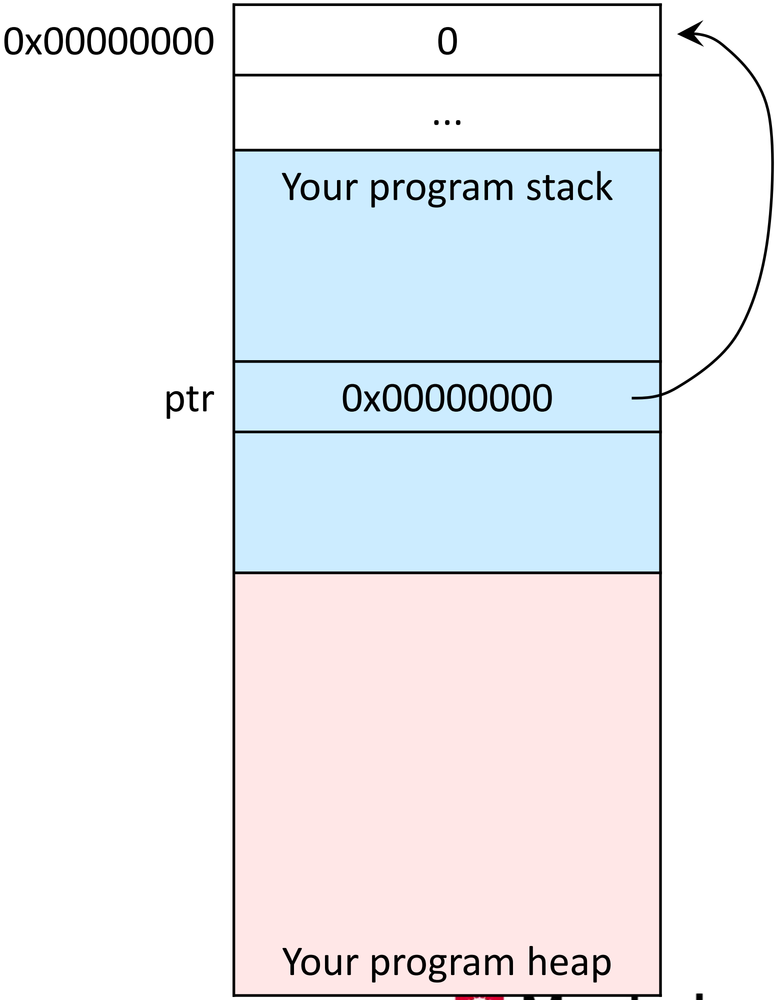
```
int *ptr;
```



Declaring Pointers Safely

- To prevent accidental alteration of anything important, pointers should always be initialised to NULL.
- The zeroth memory location of RAM is kept empty for this reason. [1]

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



0x00000000 | 0

...

Your program stack

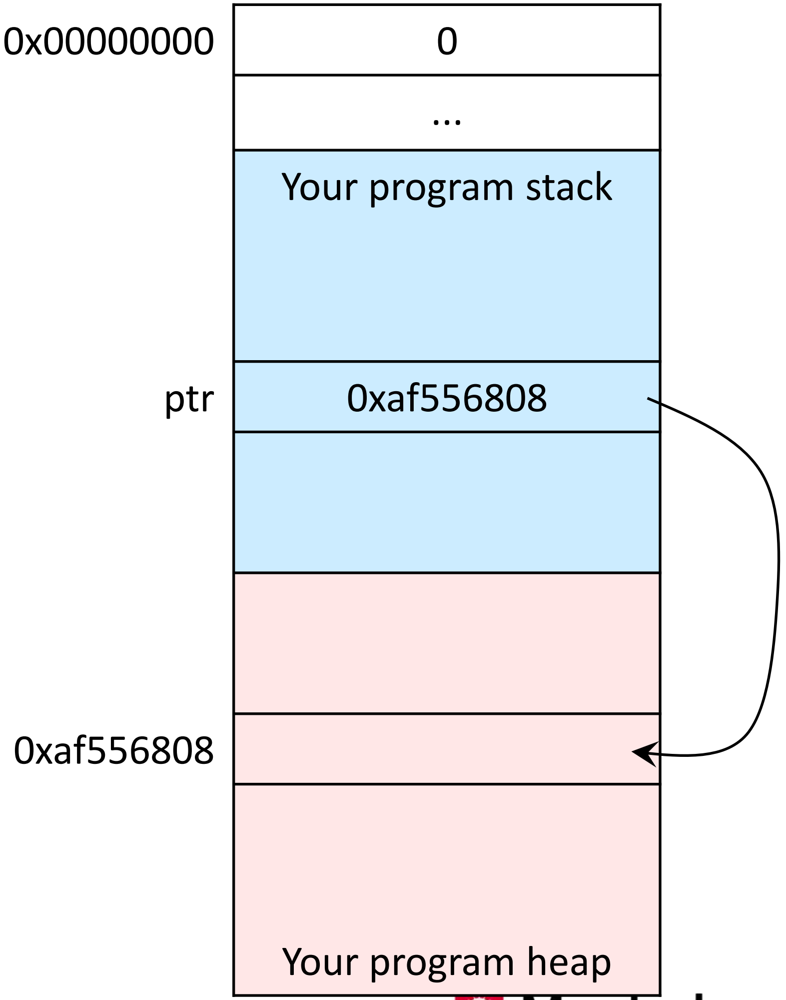
ptr | 0x00000000

Your program heap

Allocating Memory

- The allocation of memory is then done with the **`new`** keyword.
- This allocates a memory location on the heap of the given size (in this case an int).

```
int *ptr = NULL;

ptr = new int; [1]
```



0x00000000 | 0

...

Your program stack

ptr | 0xaf556808

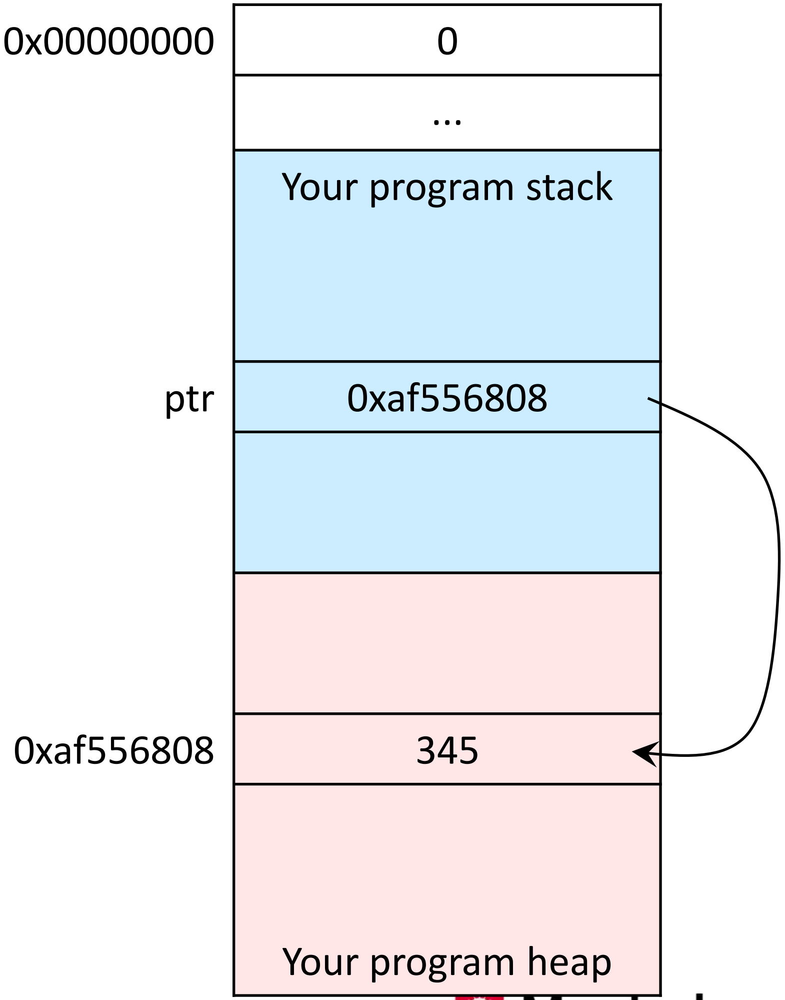
0xaf556808

Your program heap

Using Pointers

- To place a value in the memory location, the * dereferencing operator is used. [1]

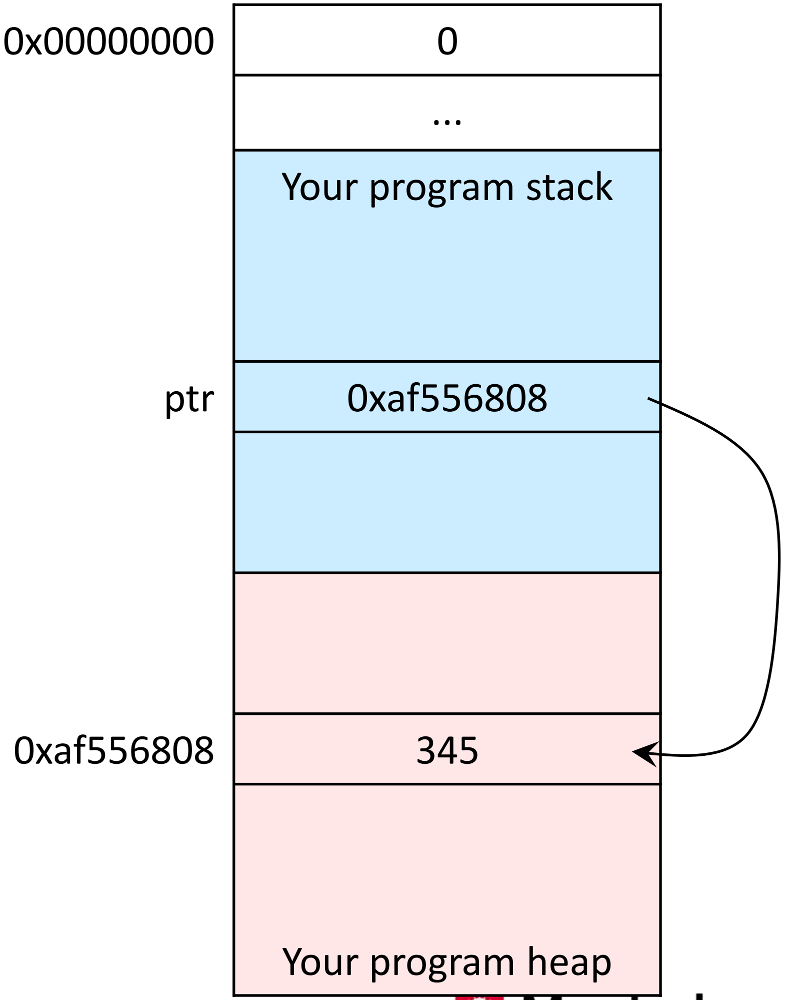
```
int *ptr = NULL;
ptr = new int;
*ptr = 345;
```



Using Pointers

- This is also used for accessing the location for output etc.

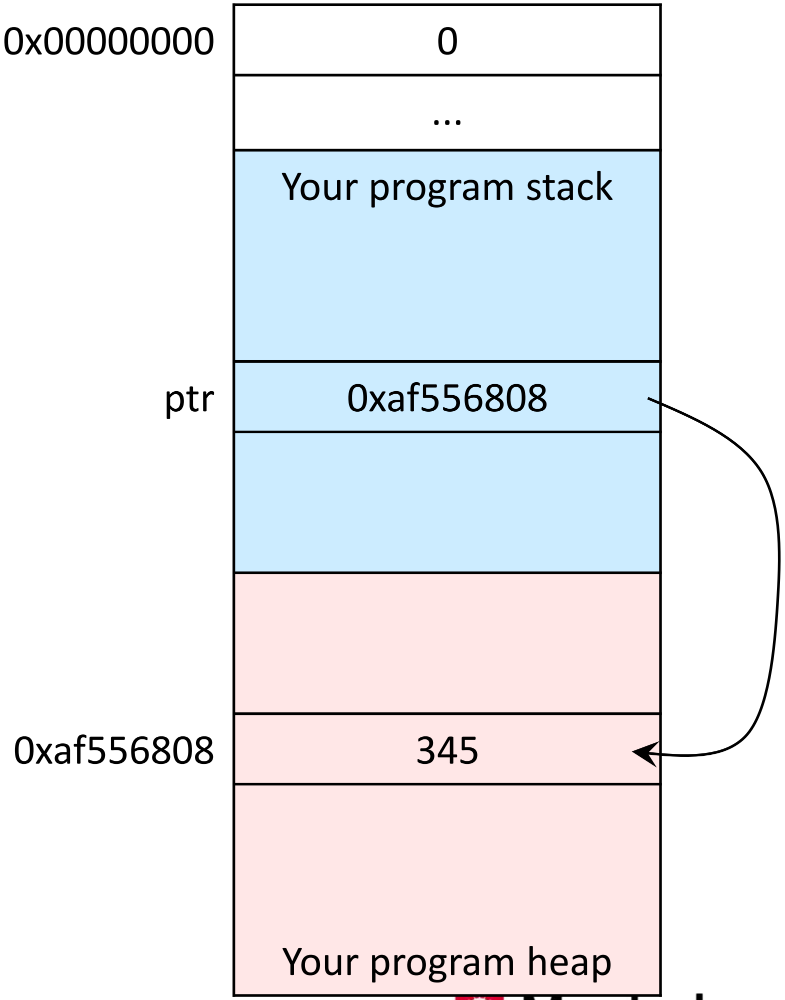
```
int *ptr = NULL;
ptr = new int;
*ptr = 345;
cout << *ptr << endl;
```



Where is the Pointer Pointing?

- You could also output the location of the memory being used, rather than the contents of the memory location.

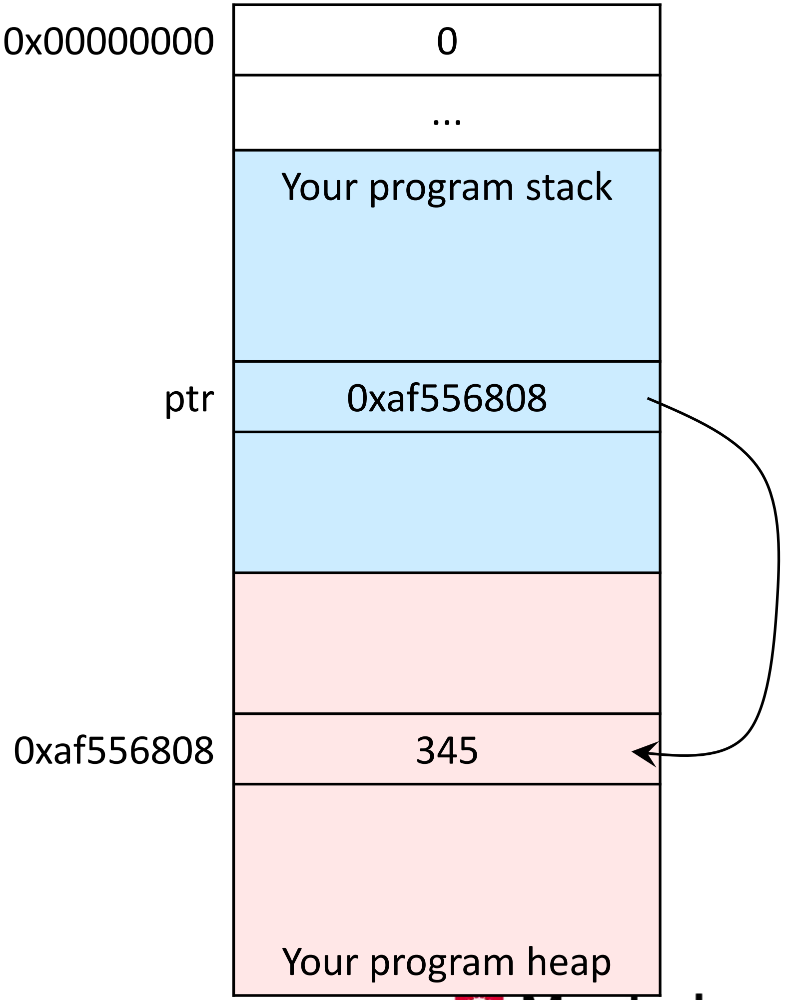
```
int *ptr = NULL;
ptr = new int;
*ptr = 345;
cout << ptr << endl;
```



Releasing Memory

- Every **`new`** must have a matching **`delete`**, so that memory is released back to the OS.

```
int *ptr = NULL;
ptr = new int;
...
delete ptr;
```



0x00000000 0

...

Your program stack

ptr 0xaf556808

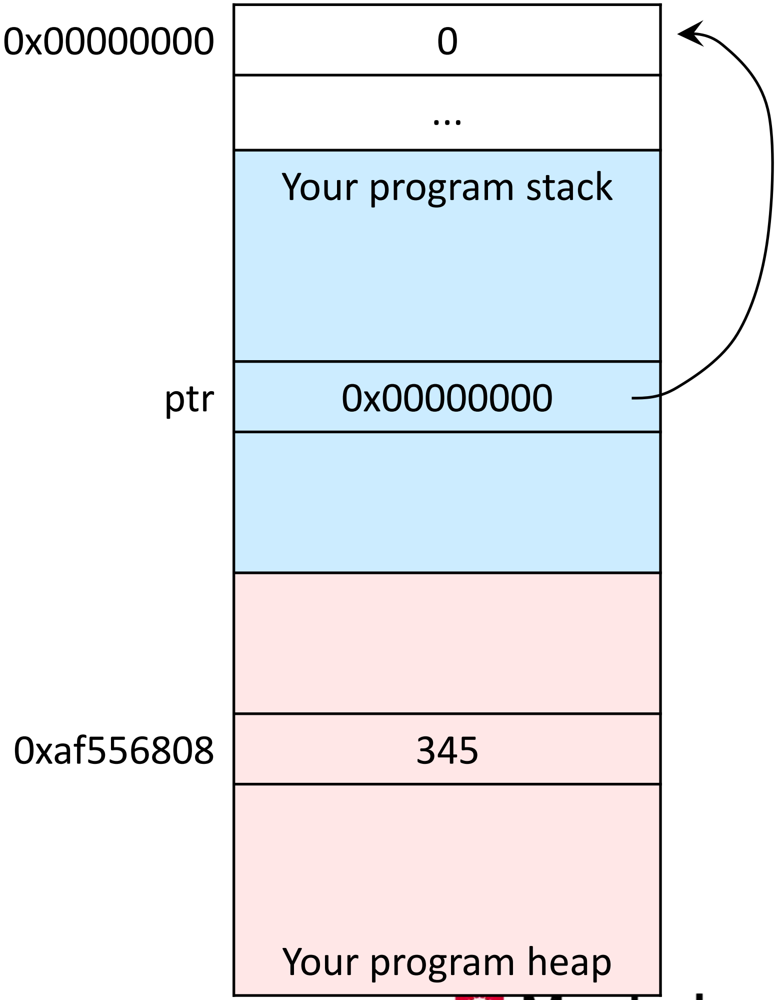
0xaf556808 345

Your program heap

Releasing Memory Safely

- Followed, of course, by reassigning the pointer to NULL, so that it does not point to memory over which it no longer should have control.

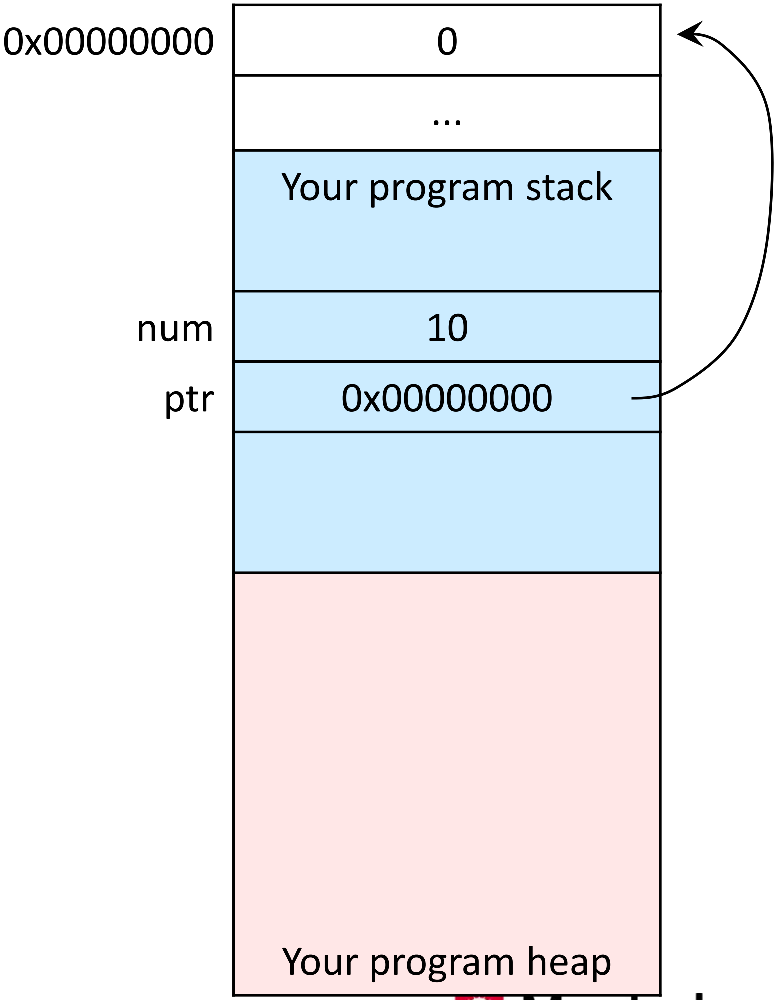
```
int *ptr = NULL;
ptr = new int;
...
delete ptr;
ptr = NULL; // to be safe
```



Pointing at Other Variables

- Pointers can also point at other variables:

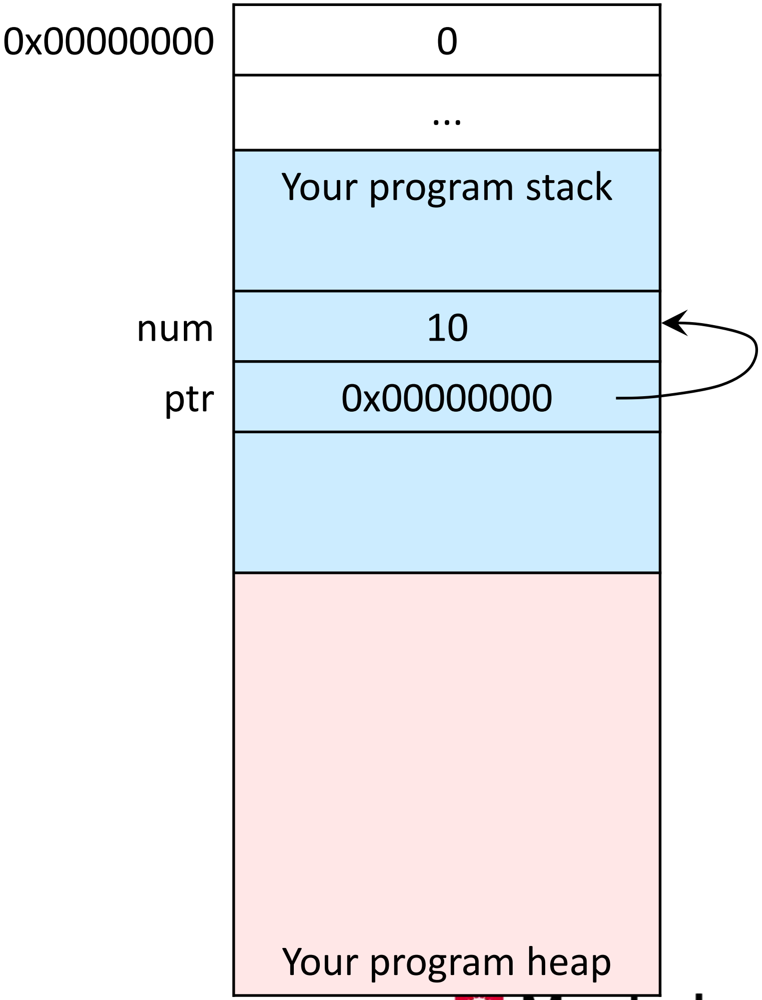
```
int *ptr = NULL;

int num = 10;
```



Pointing at Other Variables

- Pointers can also point at other variables.

```
int *ptr = NULL;

int num = 10;

ptr = &num;
```



0x00000000 | 0

...

Your program stack

num | 10

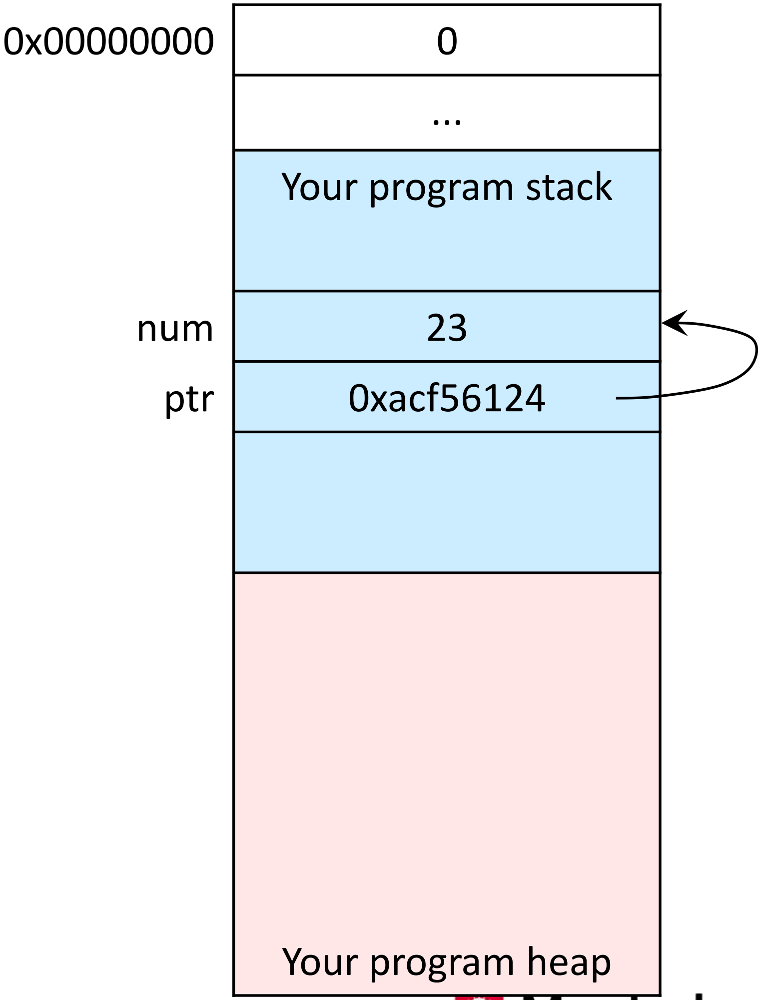
ptr | 0x00000000

Your program heap

Pointing at Other Variables

- And change their value.

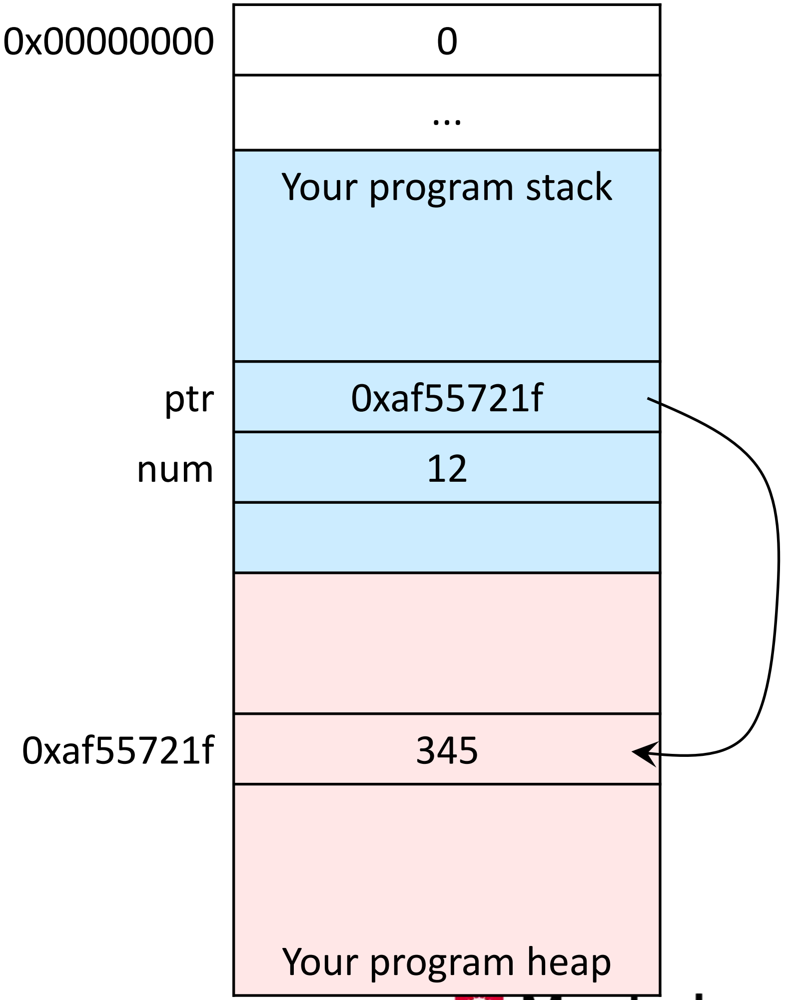
```
int *ptr = NULL;
int num = 10;
ptr = &num;
*ptr = 23;
```



Memory Leaks

- However care must be taken not to cause a memory leak.

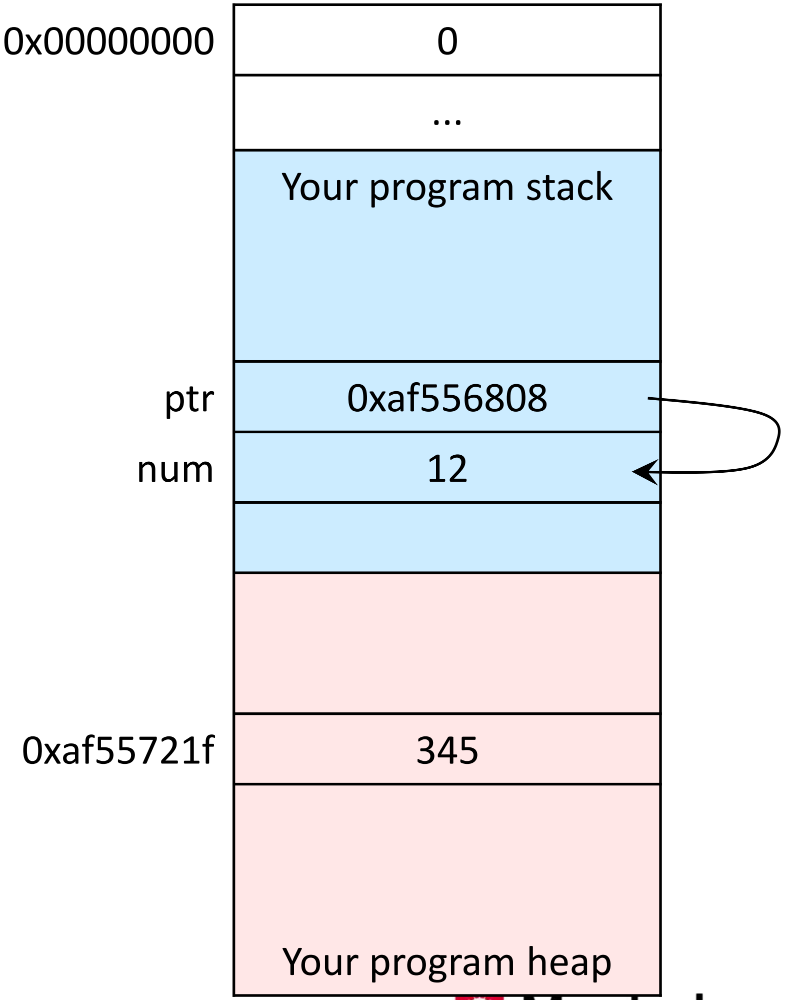
```
int num = 12;
int *ptr = new int;
*ptr = 345;
```



Memory Leaks

- However care must be taken not to cause a memory leak.

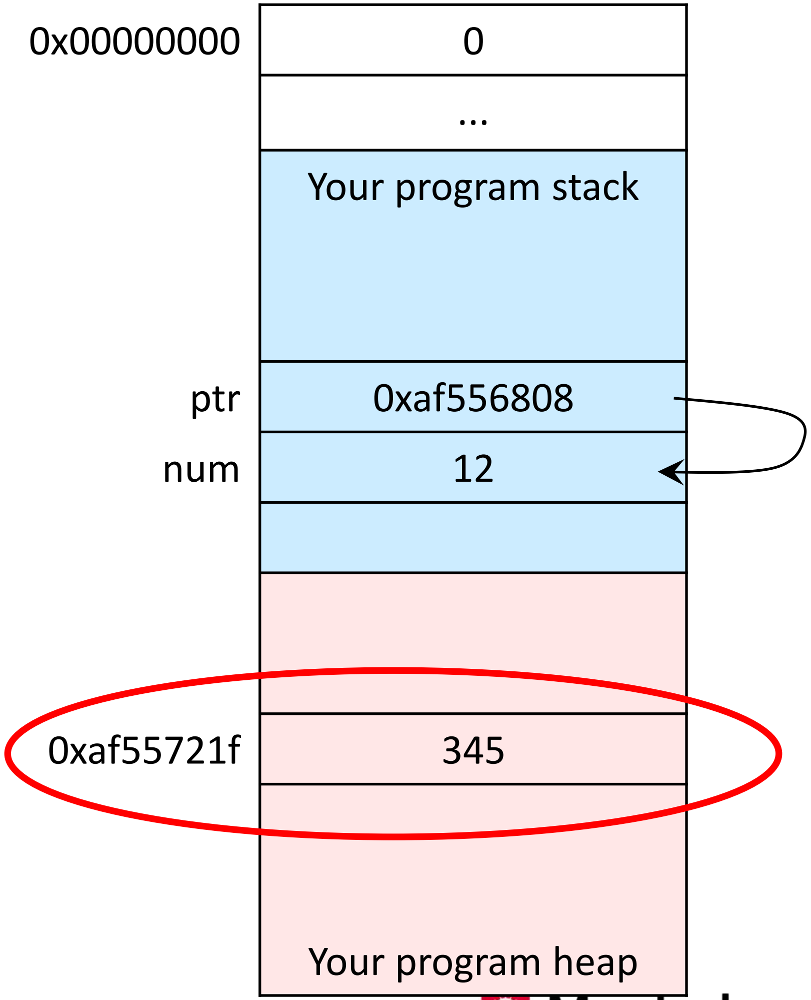
```
int num = 12;
int *ptr = new int;
*ptr = 345;
ptr = &num;
```



Memory Leaks

- The circled memory location is ‘lost’ until the program ends.

```
int num = 12;
int *ptr = new int;
*ptr = 345;
ptr = &num;
```



0x00000000 0

…

Your program stack

ptr 0xaf556808

num 12

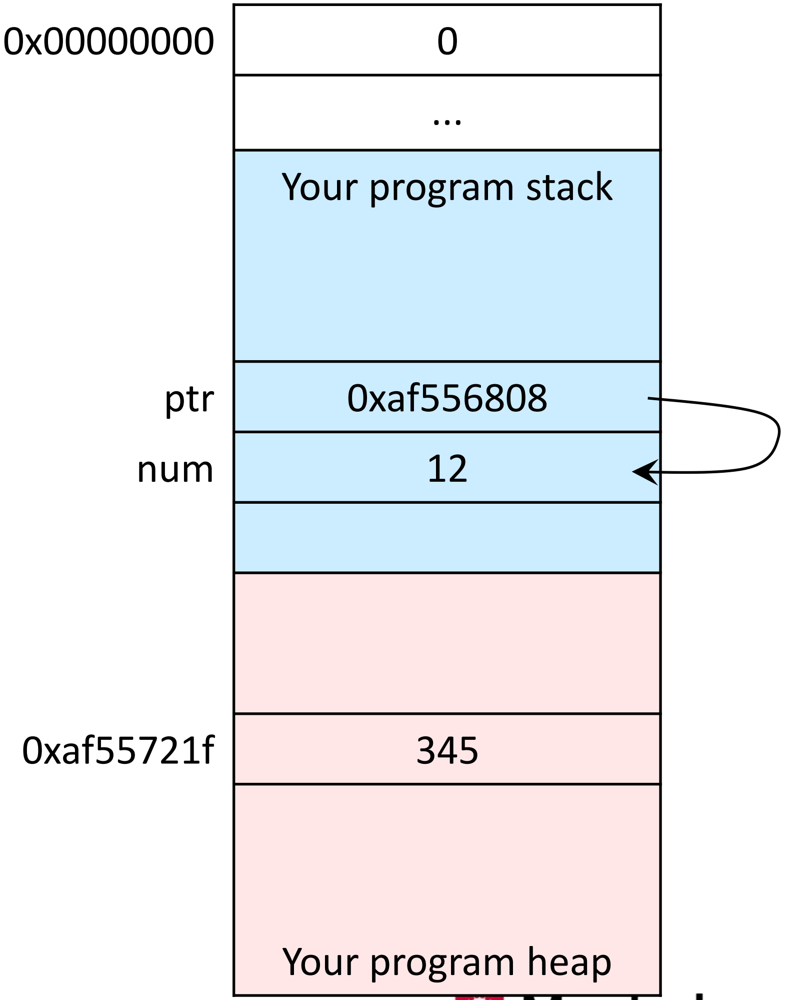
0xaf55721f 345

Your program heap

Avoiding Memory Leaks

- Of course, there should have been a delete between the last two lines of code.
- This releases the memory back to the OS again.

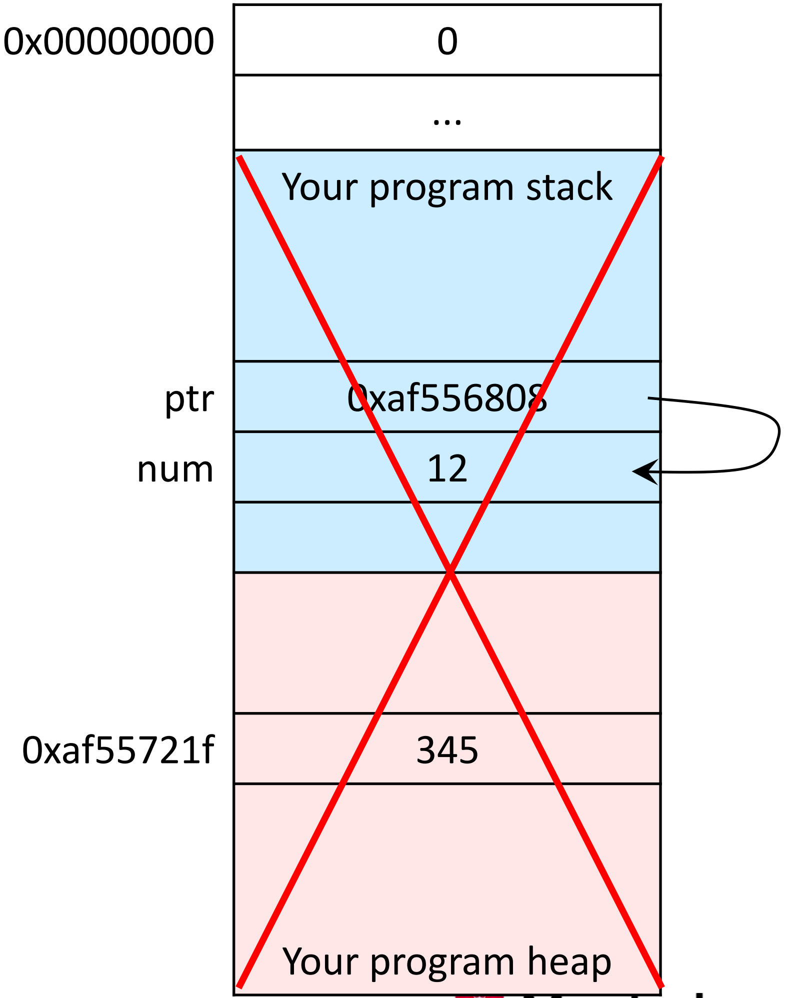
```
int num = 12;
int *ptr = new int;
*ptr = 345;
delete ptr;
ptr = &num;
```



Care with `delete`

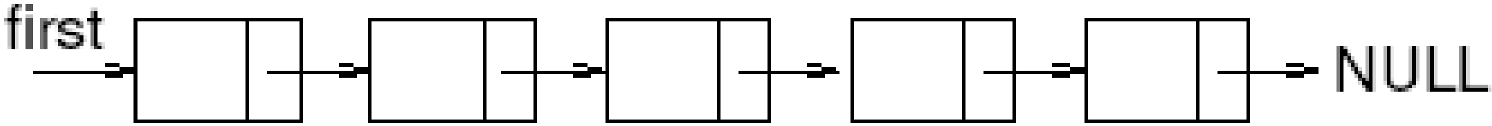
- But beware where you put it: trying to delete stack memory will cause trouble.

```
int num = 12;
int *ptr = new int;
*ptr = 345;
ptr = &num; //num not on heap
delete ptr; // oops
```



Uses of Pointers

- There are two main uses for pointers.
- The first is for array and string access.
- The second is where pointers are used to create lists or trees: data structures where the next piece of data can only be found by traversing a link from the last piece:



Malik: Chapter on Pointer - exercises

- 2. Given the declaration:

- int x;
- int *p;
- int *q;

- Mark the following statements as valid or invalid.
- If a statement is invalid, explain why:

- a. p = q;
- b. *p = 56;
- c. p = x;
- d. *p = *q;
- e. q = &x;
- f. *p = q;

- 3. What is the output of the following C++ code?

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

*p = 35;
*q =  98;
*p  = *q;

cout « x «  "     " « y « endl;
cout « *p «   "     "  « *q «   endl;
```

- 4. What is the output of the following C++ code?

- `int x;`
- `int y;`
- `int *p = &x;`
- `int *q = &y;`
- `x = 35;`
- `y =  46;`
- `p = q;`
- `*p = 18;`
- `cout « x «  "     " « y « endl;`
- `cout « *p «   "     "  «  *q «  endl;`

Readings

- Textbook (by Malik): Chapter on Pointers, Classes, .. etc. See subsections on Pointer Data Type and Pointer Variables; Address of operator; Dereferencing operator. A different edition may have a different chapter number and pages.
- 101 Coding Standards: **Rules 51 and 52**. It is one of the important references (see unit outline) we use in the unit. See 101 C++ Coding Standards **online** resource. [1]
- Watch the videos on pointers “Video Lecture on Pointers.htm” Function pointers are covered when we cover the tree data structure later on.
- Find out about “**RAII**”. Why is the RAII concept so important that you should not violate it?

Murdoch
UNIVERSITY

Videos

- Pointers - Stanford University

  https://www.youtube.com/watch?v=H4MQXBF6FN4

- Bits and bytes; floating point representation - Stanford University

  https://www.youtube.com/watch?v=jTSvthW34GU

- How pointers get used; usage of void pointers

  https://www.youtube.com/watch?v=_eR4rxnM7Lc

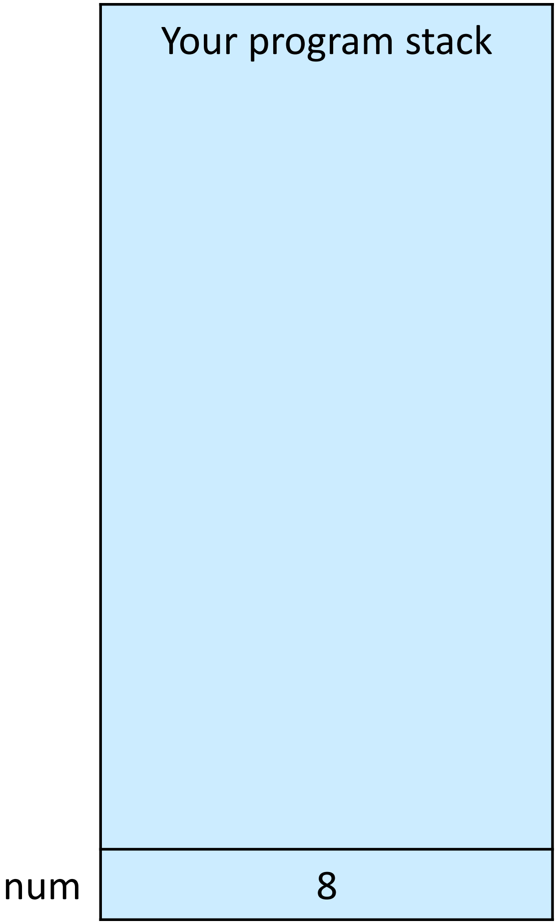
Parameters

- Parameters can be passed in four main ways.
  - by value
  - by reference
  - by constant reference
  - by pointer

Value Parameters

```
void Func1 (int number);
...


int num = 8;
```

`number` is a value parameter

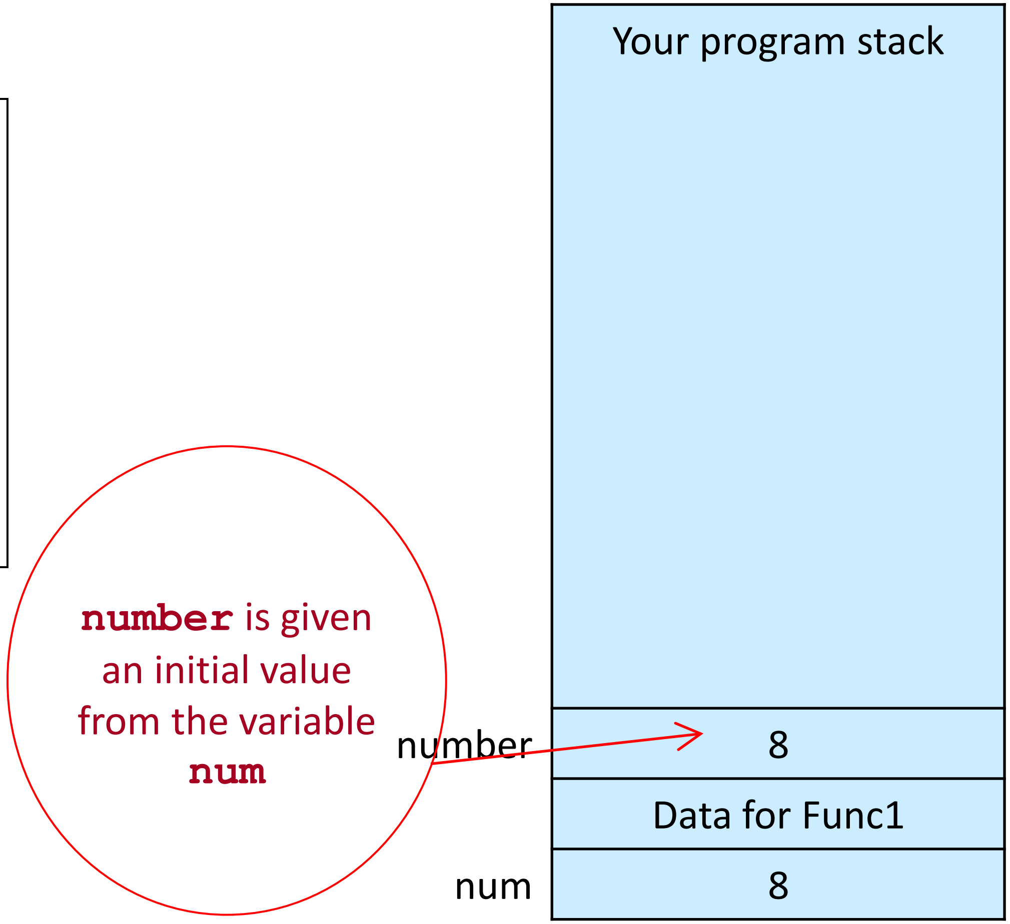


Your program stack

num 8

Value Parameters

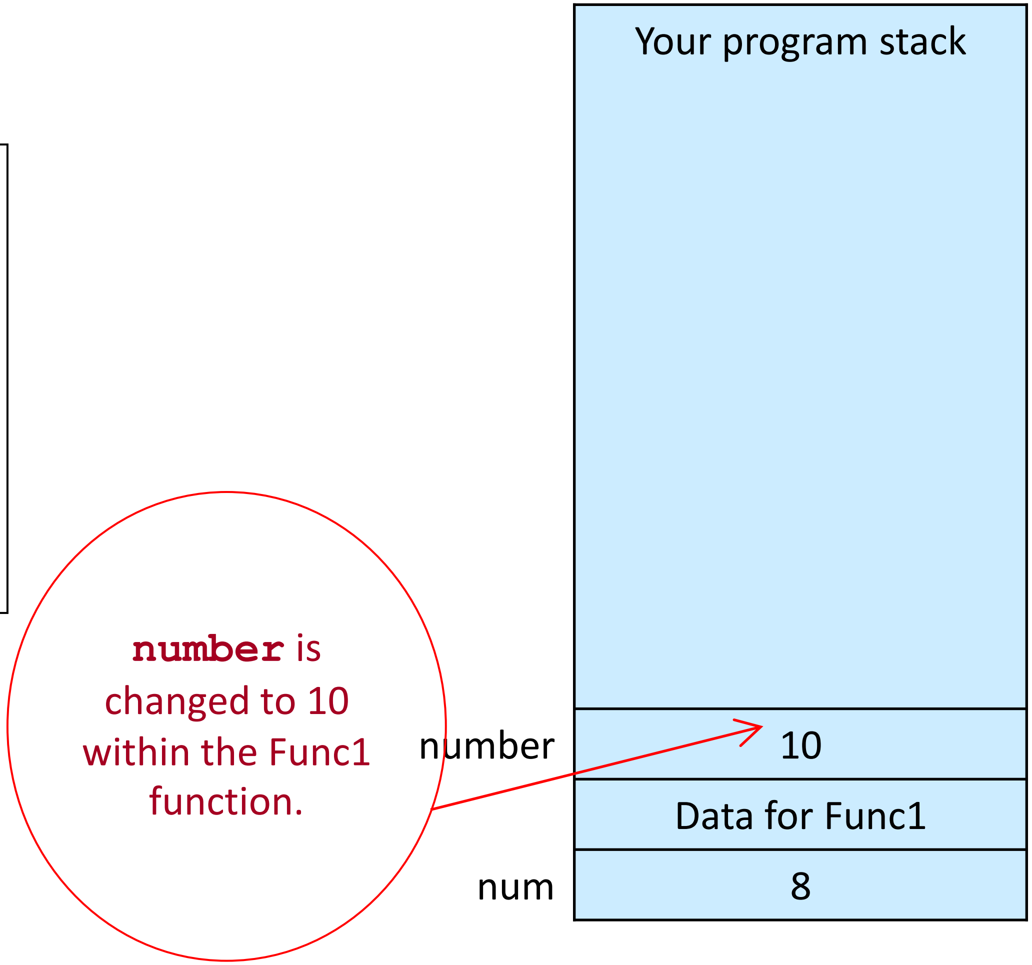
```
void Func1 (int number);
...


int num = 8;
Func1 (num);
```



number is given an initial value from the variable **num**

Value Parameters

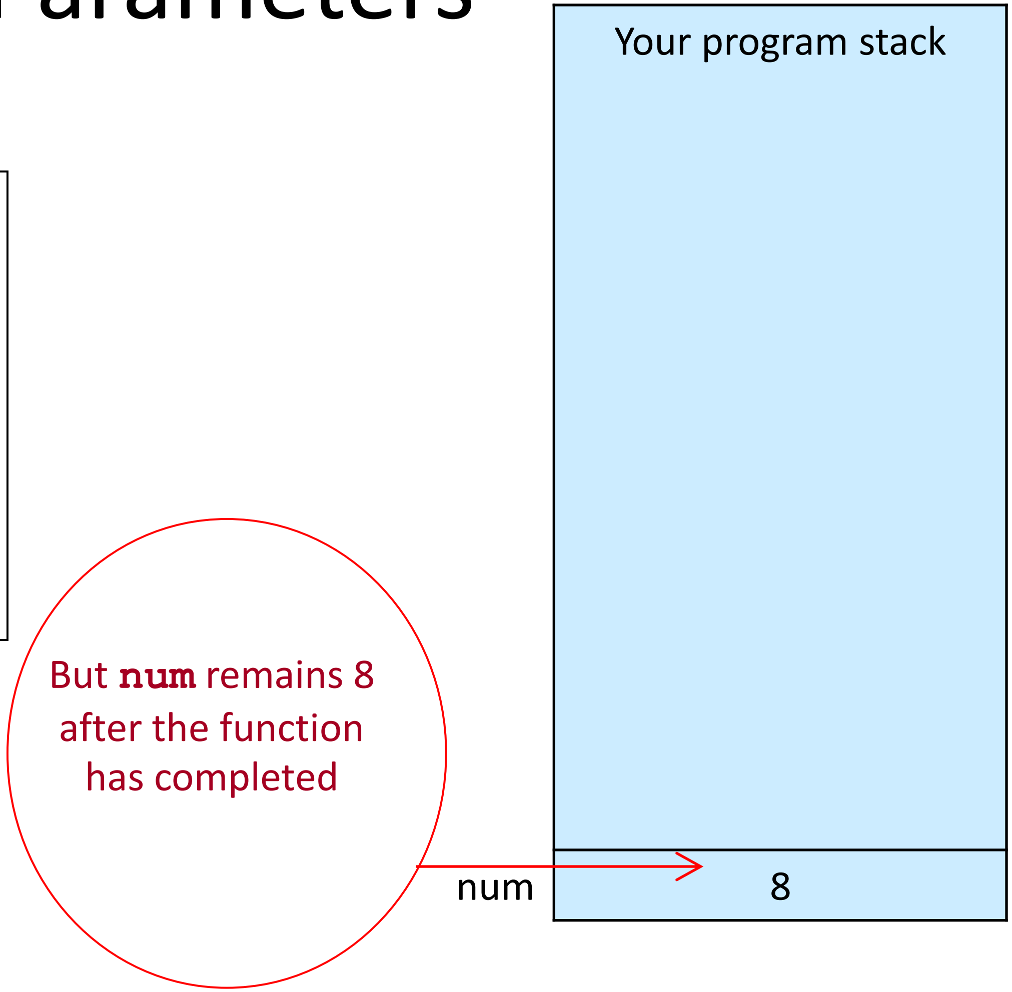
```
void Func1 (int number);
...


int num = 8;
Func1 (num);
```



number is changed to 10 within the Func1 function.

Value Parameters

```
void Func1 (int number);
...


int num = 8;
Func1 (num);
```

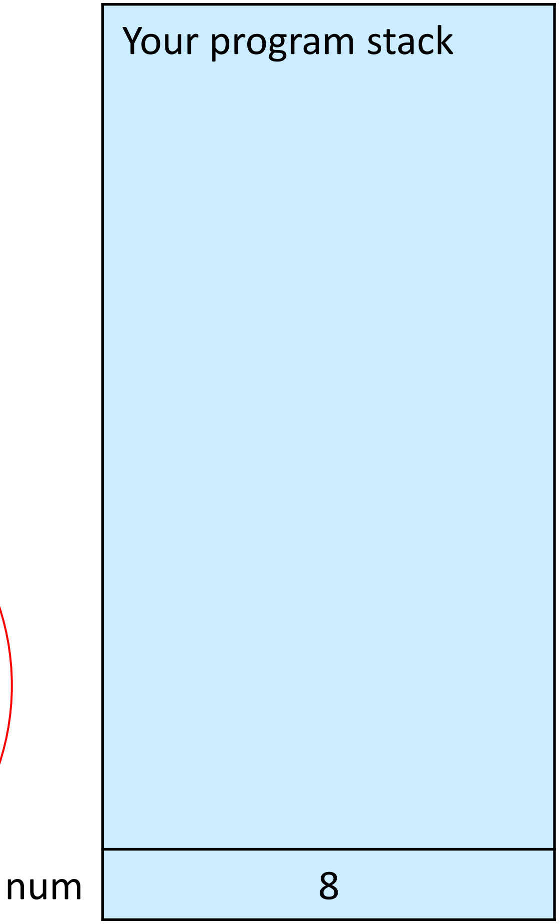


But **num** remains 8 after the function has completed

Reference Parameters

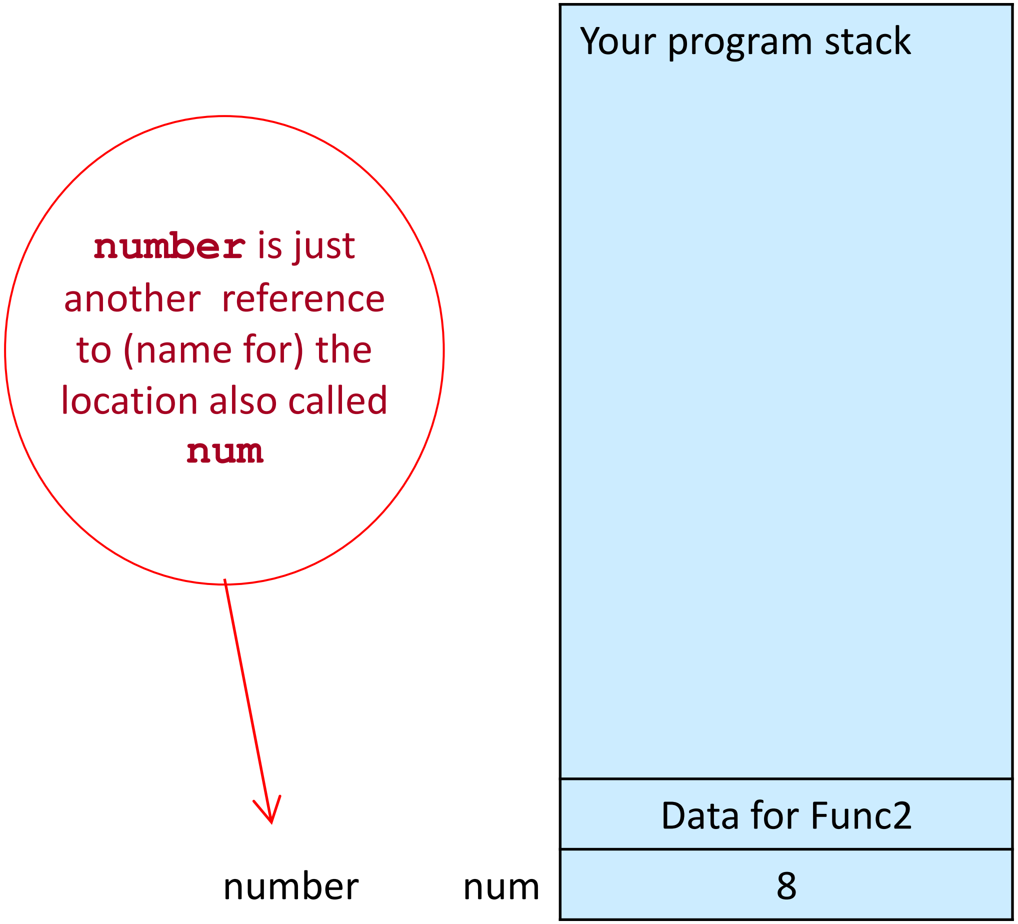
```
void Func2 (int &number); // [1]
...


int num = 8;
```

`number` is a reference parameter i.e. another name for something



Reference Parameters

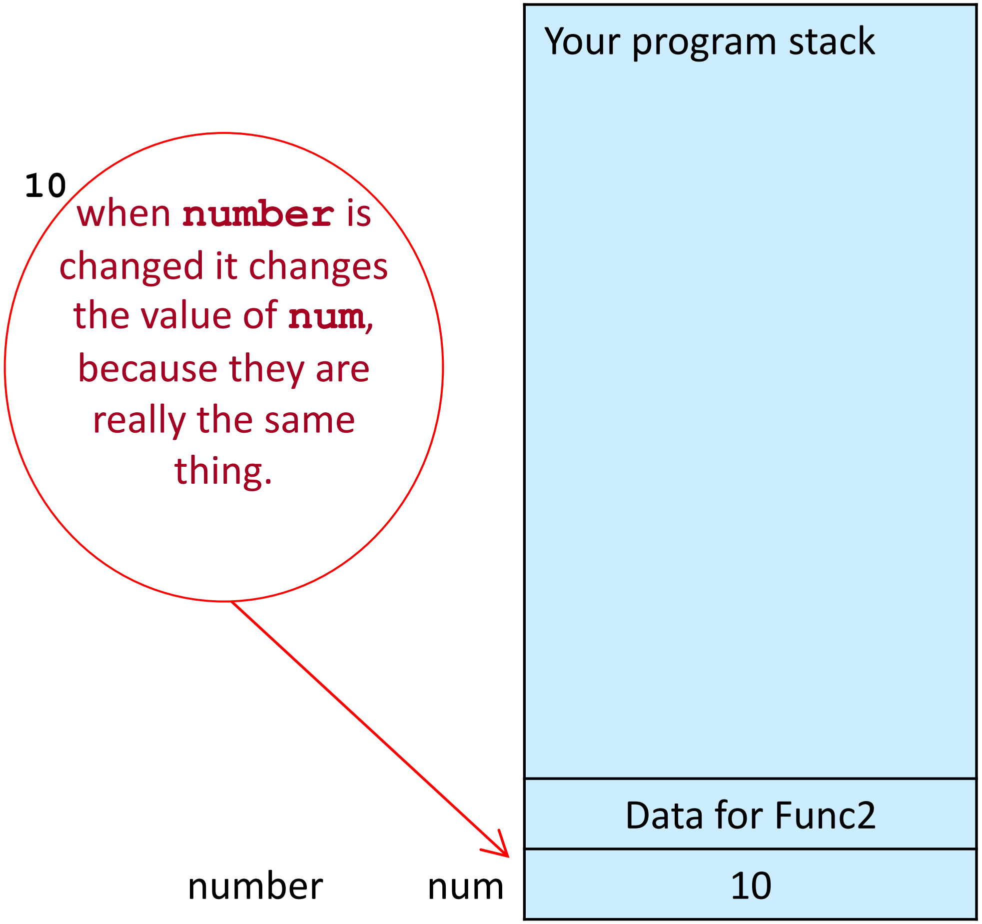
```
void Func2 (int &number);
...


int num = 8;
Func2 (num);
```



number is just another reference to (name for) the location also called **num**

Reference Parameters

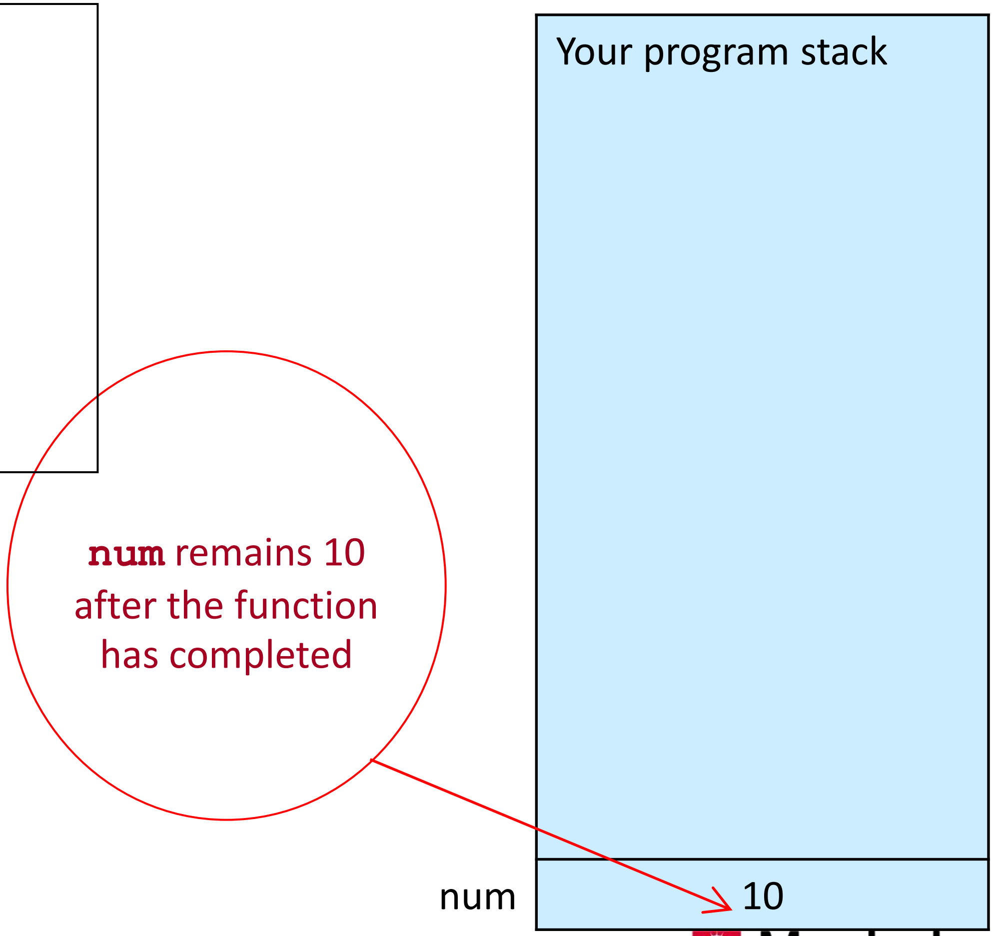
```
void Func2 (int &number){
 number = 10; //changes number to 10
}

//--------------------
int num = 8;
Func2 (num);
```



when **number** is changed it changes the value of **num**, because they are really the same thing.

Reference Parameters

```
void Func2 (int &number);
...


int num = 8;
Func2 (num);
```



Your program stack

`num` remains 10 after the function has completed

Constant Reference Parameters

```
void Func3 (const int &number);
...


int num = 8;
```

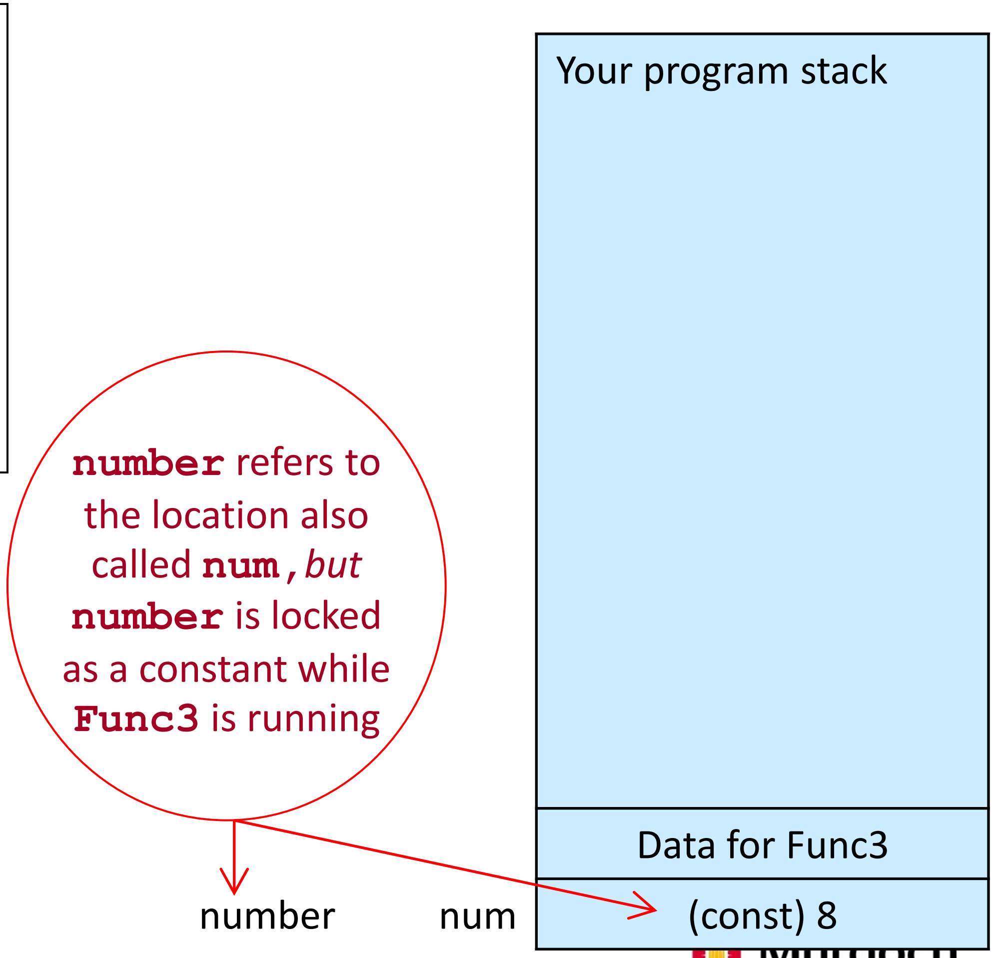
`number` is a *constant* reference parameter

Your program stack

num 8

Constant Reference Parameters

```
void Func3 (const int &number);
...


int num = 8;
Func3 (num);
```

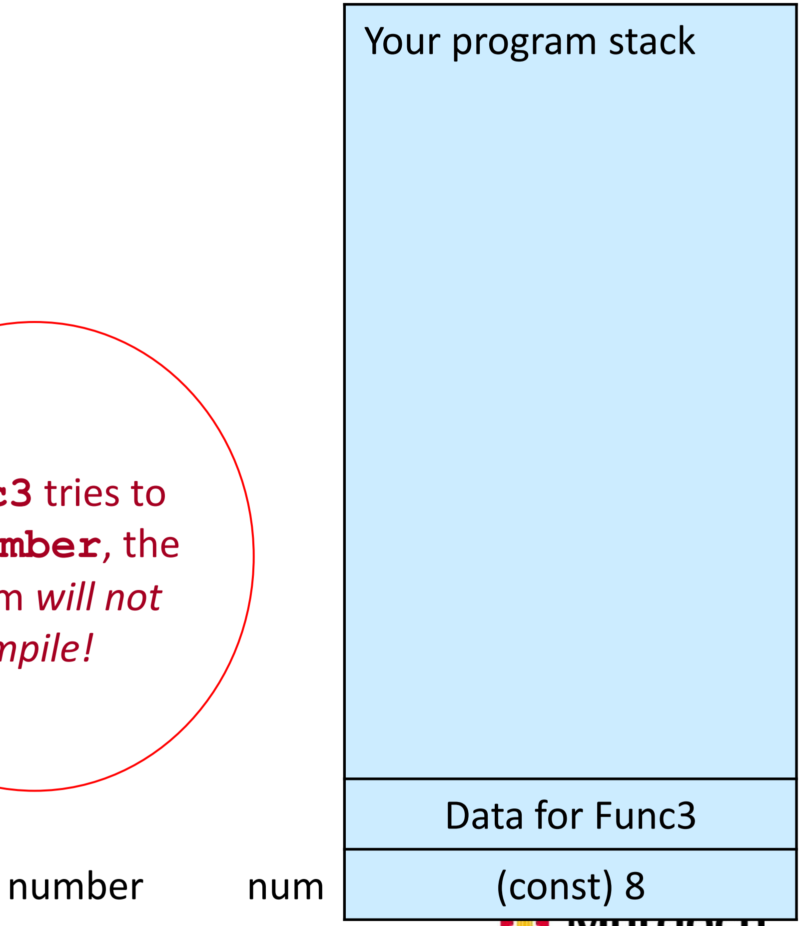


number refers to the location also called **num**, *but* **number** is locked as a constant while **Func3** is running

Constant Reference Parameters

```
void Func3 (const int &number);
...// attempt to change number
...// will not compile – good

int num = 8;
Func3 (num);
```

If **`Func3`** tries to alter **`number`**, the program *will not compile!*



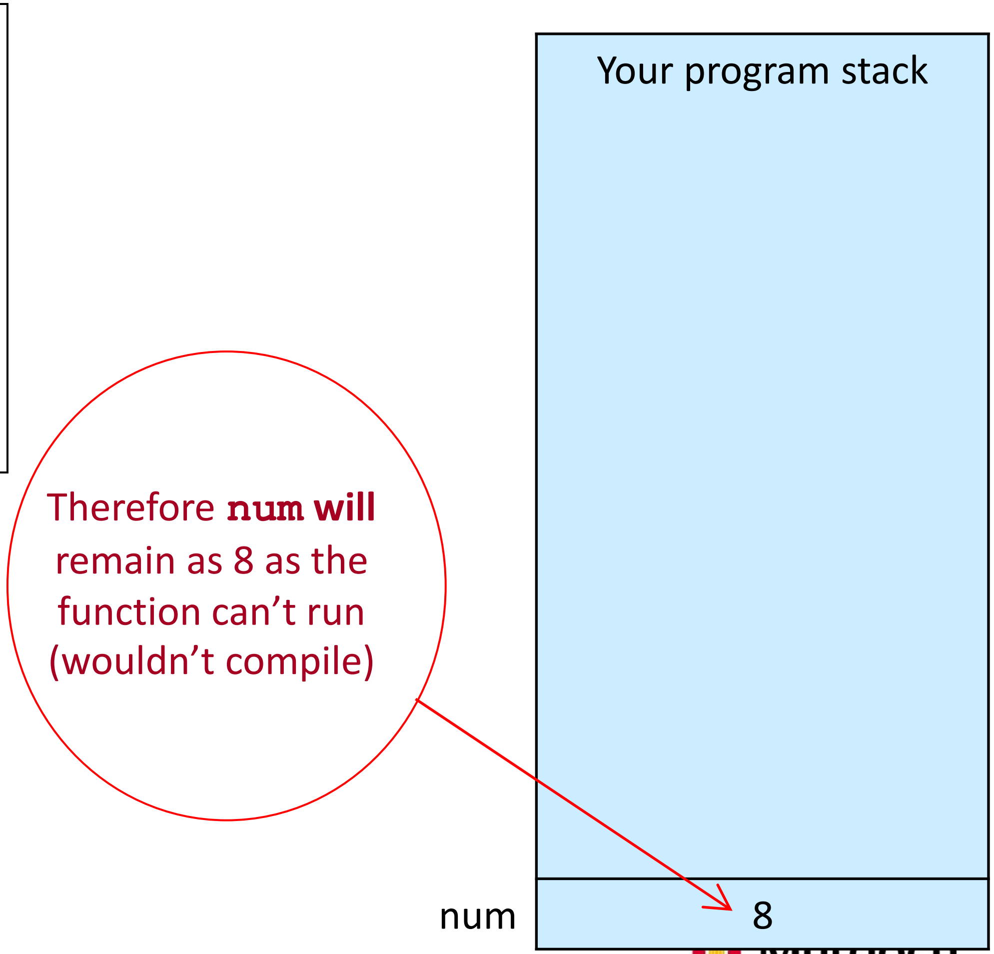
Your program stack

Data for Func3

number num (const) 8

Constant Reference Parameters

```
void Func3 (const int &number);
...


int num = 8;
Func3 (num);
```



Therefore **num will** remain as 8 as the function can’t run (wouldn’t compile)

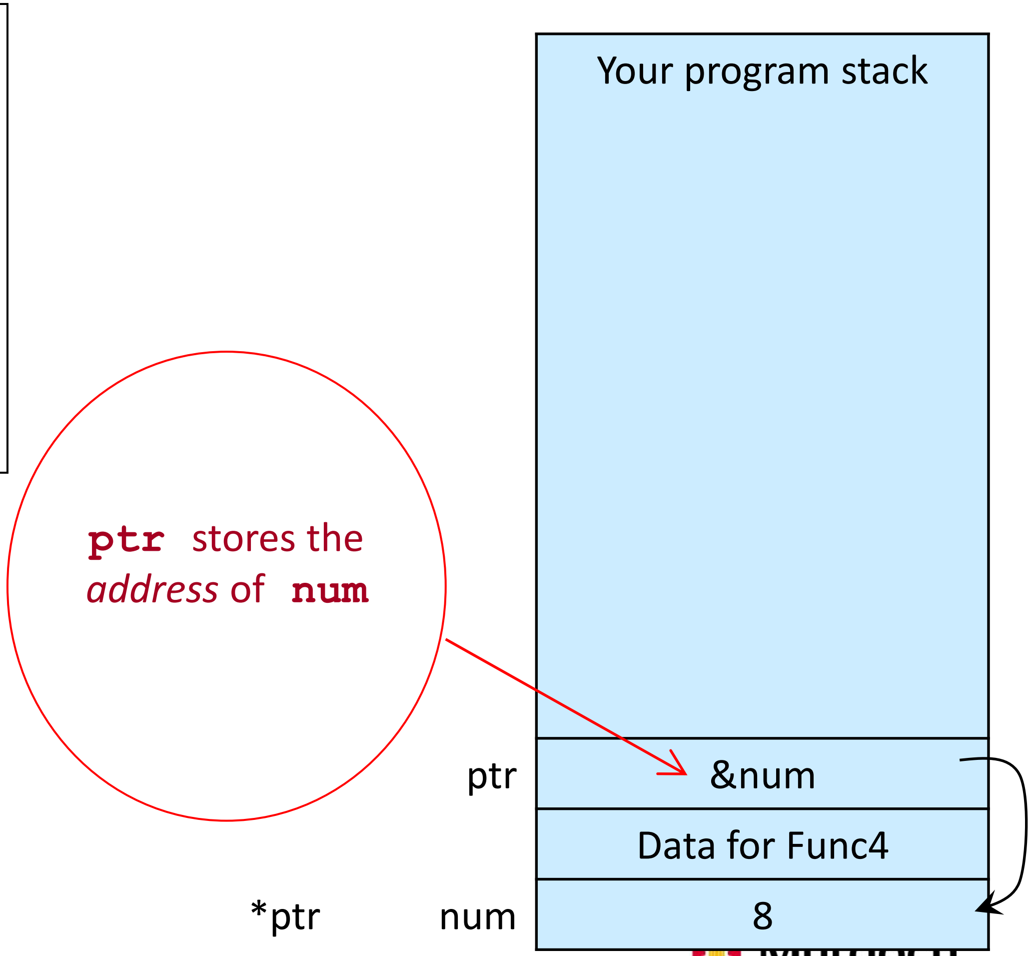
Pointer Parameters

```
void Func4 (int *ptr);
...


int num = 8;
```

`ptr` is a pointer parameter, therefore within **`Func4`**, **`ptr`** is a pointer *not* an integer

Pointer Parameters

```
void Func4 (int *ptr);
...


int num = 8;
Func4 (&num);
```



`ptr` stores the *address* of **`num`**

Your program stack

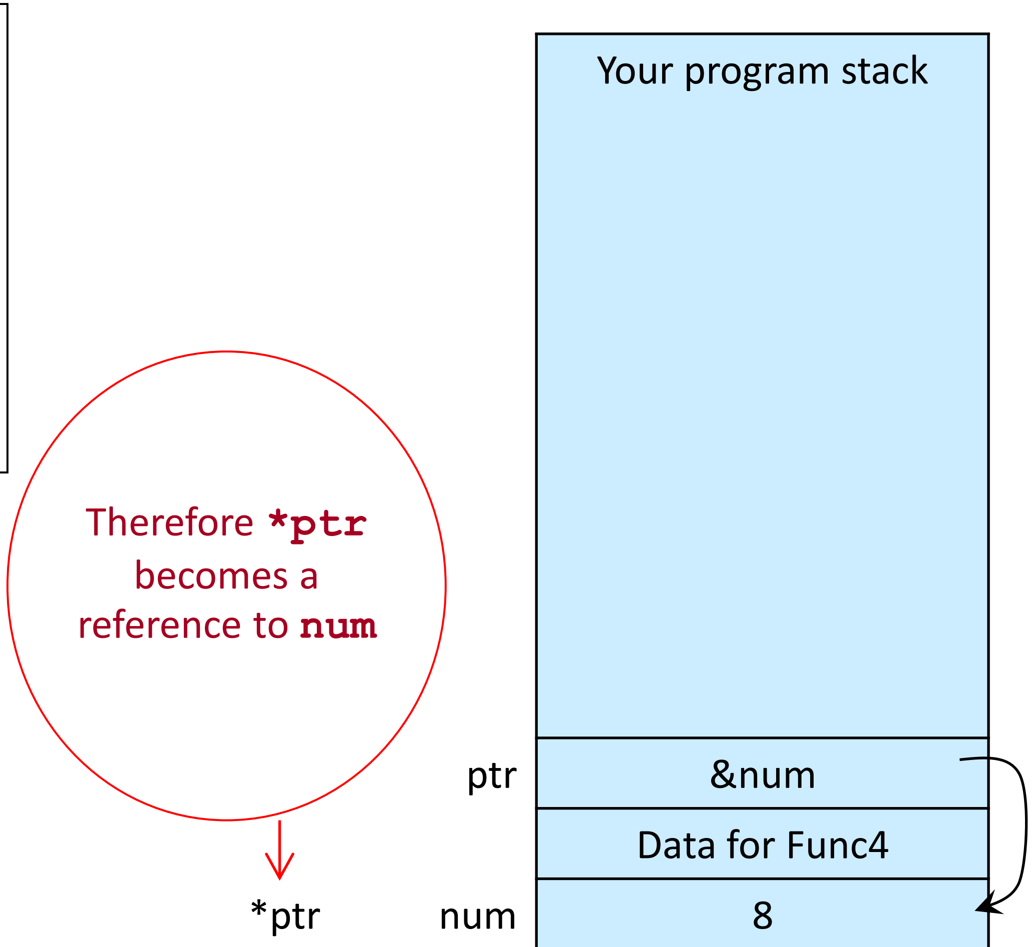
ptr &num

Data for Func4

*ptr num 8

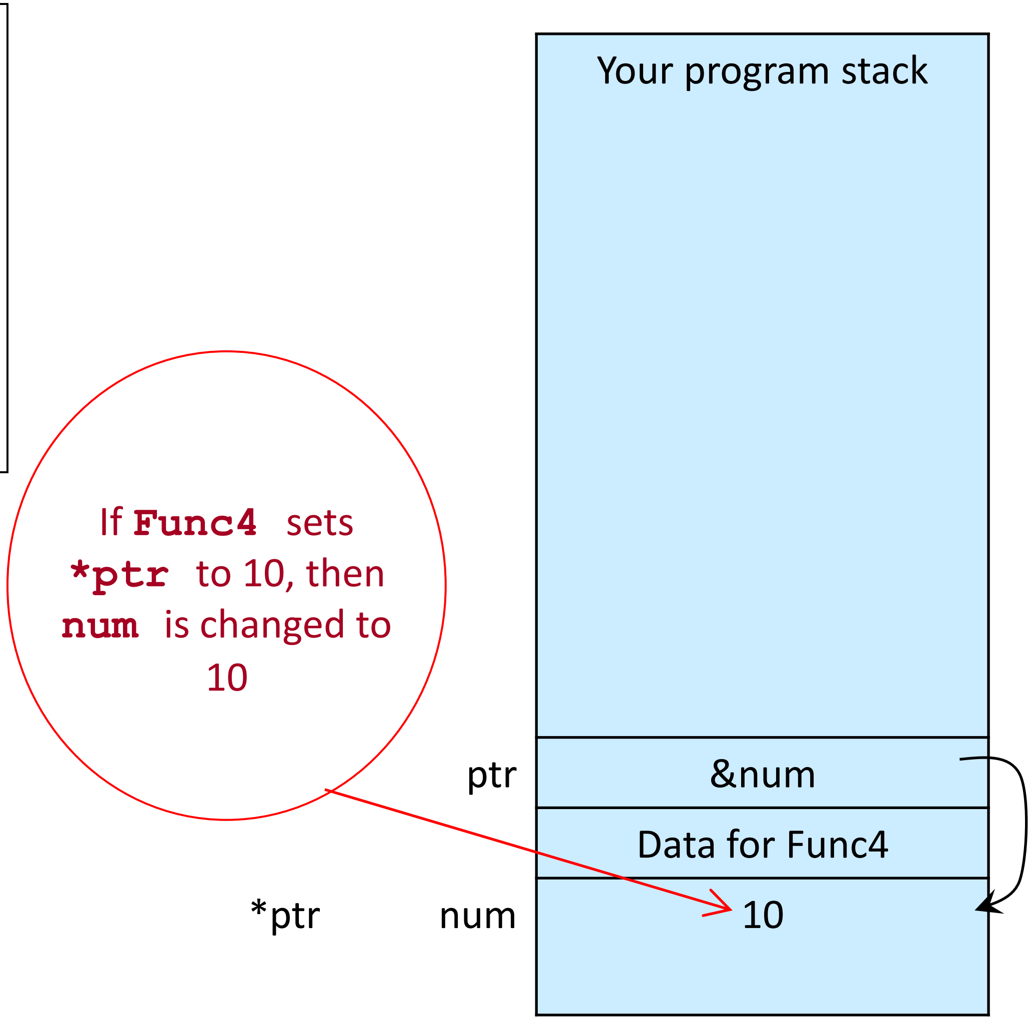
Pointer Parameters

```
void Func4 (int *ptr);
...


int num = 8;
Func4 (&num);
```


Therefore ***ptr** becomes a reference to **num**

Pointer Parameters

```
void Func4 (int *ptr);
...



int num = 8;
Func4 (&num);
```

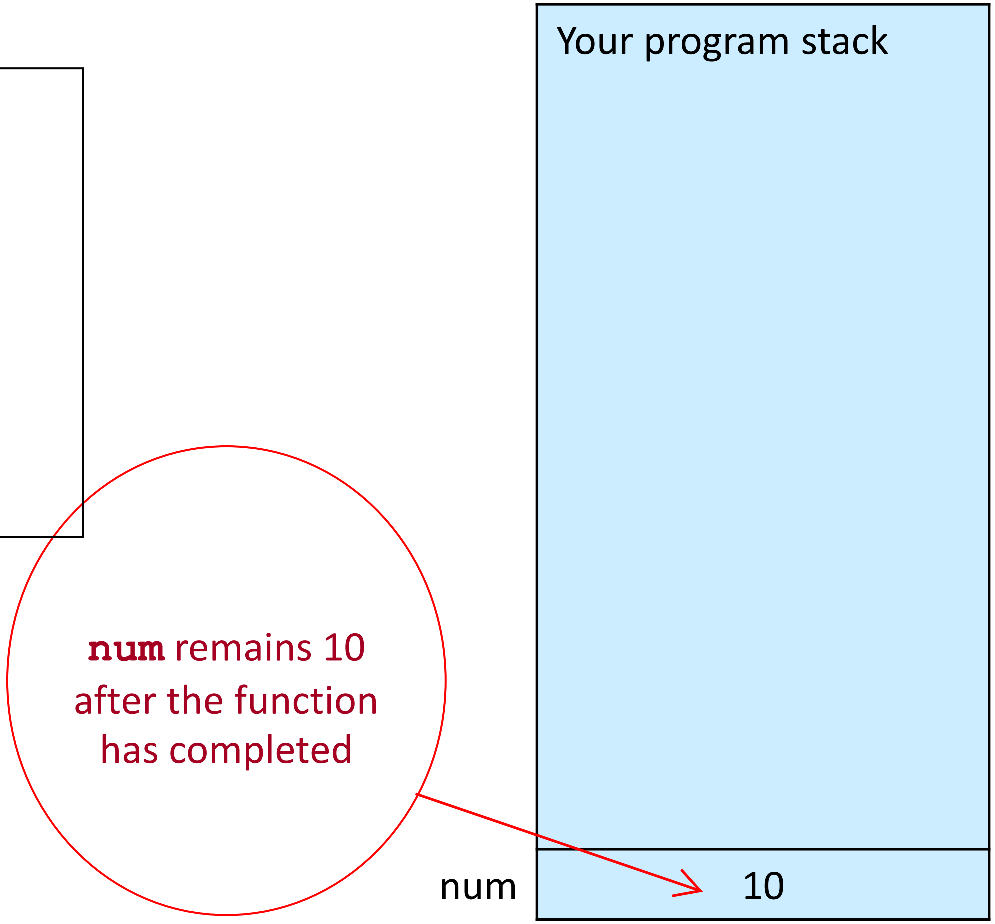


If **`Func4`** sets **`*ptr`** to 10, then **`num`** is changed to 10

Pointer Parameters

```
void Func4 (int *ptr);
...


int num = 8;
Func4 (&num);
```

- For further work (not necessary for this unit), find out about smart pointers, auto_ptr, and Opaque pointer. [1]



Pointers and References

- Find out about the following:
- It is possible to declare a pointer with no initial value? Is it possible to declare a reference which does not contain an initial value?
- A pointer variable can be changed to point to something else. Can this be done with a reference?
- A pointer can be set to contain the NULL (or **nullptr**) value. Can you make a reference NULL (or nullptr)? [1]
- Can you do pointer like arithmetic on references?

Readings

- Chapter(s) on User-Defined Functions, section on Value Returning Functions; section on Reference variables as parameters.

- If you are using another edition of the textbook, look up the chapter title and section number in the contents page.