# CS-202

# C++ Functions

# Pointers & References

**C. Papachristos**
Autonomous Robots Lab
University of Nevada, Reno

# Course Week

## Course , Projects , Labs:

| Monday | Tuesday | Wednesday | Thursday | Friday |
|---|---|---|---|---|
| | | | Lab (8 Sections) | |
| | CLASS | | CLASS | |
| PASS Session | PASS Session | **Project DEADLINE** | NEW Project | |

Your 1st Project Deadline is this Wednesday 1/30.

- PASS Sessions TBD, once established use them to get all the help you need!
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Past that, NO Project accepted. Better send what you have in time!

# Today's Topics

## C++ Functions

- Parts
  - Prototype
  - Definition
  - Call
- Return
- Parameters / Arguments
- Libraries

## Pointers

## References

## Function Parts

Function Declaration (Prototype)

- Information for compiler to properly interpret calls.

Function Definition (Implementation)

- Actual implementation (i.e., code) for function.

Function Call

- How function is actually used by program.
- Transfers execution control to the function.

# Functions

## **Function Declaration** (Prototype)

Gives compiler information about the function

➢ How to interpret calls to the function.

```
<return type> <function_name> (<parameter_types>);
double sprintSum (int, double, char []);
```
or
```
double sprintSum (int n, double d, char s[]);
```

Forward Declaration : Semicolon-terminated

Must list the parameters' data types (at least).

Placed before any calls

In declaration space of `main()`.

Or above `main()` for global access.

## **Function Definition** (Implementation)

Definition of the function:

```
double printSum (int n, double d, char s[]) {
    double sum = d + n;
    sprintf(s, "%.3f", sum);
    return sum;
}
```

Function Block : Brace-enclosed

- Function definition must match prototype.
  Placed AFTER the function `main()` (NOT inside).
- All definitions of equal order, no function *needs* to be contained inside another.

- Function name, parameter(s) type, and return type all must match the prototype's.
- `return` statement sends data back to the caller.

## Function Call

Much like a standard C call:

```
char printSum[256];
double returnSum = sprintSum(10, 0.1, printSum);
```

- Returns a `double`.
  (Assigned to variable `returnSum`)

Arguments:

- The literals `10, 0.01`.
  (Can also pass variables – do they have to be `int` and `double`?)
- A `char` array variable `printSum`
  (Has to match `char []` type – formally `char *`.)

# Functions

## Function Parts

- Function Prototype
- Function Definition
- Function Call

```cpp
#include <iostream>
using namespace std;

double totalCost(int numberParameter, double priceParameter);
//Computes the total cost, including 5% sales tax,
//on numberParameter items at a cost of priceParameter each.

int main( )
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = totalCost(number, price);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}

double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal*TAXRATE);
}
```

*Function declaration; also called the function prototype*

*Function call*

*Function head*

*Function body*

*Function definition*

## `return` Statement(s)

Transfers control back to the calling function.

Special case: "`void`" functions:

No value back, Functions that only have side effects (e.g., print out information).

Similar declaration to "regular" functions

```
void printResults(double cost, double tax);
```

Optional `return` statement (all other return types must have a return statement).

Typically the last statement in the definition.

Can also have multiple `return` statements.

- Transfers control *early*, (anything past it in the function Block is not executed).
- Can have multiple exit points in a function.

Typical use: *guard statements* ( `if(somethingWrong) return;` ).

## Function Parameters / Arguments

(Function) Parameter:

- Formal variable, as it appears in the function prototype.
- Part of the *Function Signature* (more on that later).

(Function) Argument:

- Actual value or variable.
- An expression used when making the function call.

Multiple Parameters / Arguments:

```
double precisionSum(double a, double b);
cout << precisionSum(0.1 * 1000000, 1e-3);
```

→ 100.001

## Function Parameters / Arguments

Variadic Functions & Arguments :

```
double precisionMultiSum(int numargs, ...);
```

Example Calls:

```
precisionMultiSum(2, 1.0, 2.0);
precisionMultiSum(3, 0.1, 0.2, 0.3);
```

```
#include <iostream>
#include <cstdarg>

double precisionMultiSum(int numargs, ...){
 double sum = 0;
 va_list ap;
  va_start(ap, numargs);
  for (int i=0; i<numargs; ++i)
    sum += va_arg(ap, double);
  va_end(ap);
  return sum;
}
int main() {
  std::cout << precisionMultiSum(2, 5.0, 2.0);
  return 0;
}
```

Actually

*Preprocessor Macros*

are used to reinterpret parameter: `...`

in expressions where `precisionMultiSum` appears.

What if ?

```
precisionMultiSum(2, 5, 2);
precisionMultiSum(2, 5.0, 2);
precisionMultiSum(2, 5, 2.0);
```

## Function Pre / Post - Conditions

Include function headers in your code.

- Contain name, pre / post – conditions:

Conditions include assumptions about program state, not just the input and output.

```
// Function name: showInterest
// Pre-condition: balance is nonnegative account
//    balance; rate is interest rate as percentage
// Post-condition: amount of interest on given
//    balance, at given rate

void showInterest(double balance, double rate);
```

Note:

Code Comments

```
// Single-line Comment Here
```

or

```
/* Multi-line
Comments Here */
```

## C++ Function Libraries

Full of useful functions!
Must "#include" appropriate library.

- Correspondence to "C" libraries:

  C++ <cmath> ~ <math.h> C analog
  <cstdlib> ~ <stdlib.h>
  <cstring> ~ <string.h>

- Console-File I/O:
  (e.g. std::cout, std::cin)
  <iostream>

- Many more…

| NAME | DESCRIPTION | TYPE OF ARGUMENTS | TYPE OF VALUE RETURNED | EXAMPLE | VALUE | LIBRARY HEADER |
|---|---|---|---|---|---|---|
| sqrt | Square root | double | double | sqrt(4.0) | 2.0 | cmath |
| pow | Powers | double | double | pow(2.0,3.0) | 8.0 | cmath |
| abs | Absolute value for int | int | int | abs(-7)<br>abs(7) | 7<br>7 | cstdlib |
| labs | Absolute value for long | long | long | labs(-70000)<br>labs(70000) | 70000<br>70000 | cstdlib |
| fabs | Absolute value for double | double | double | fabs(-7.5)<br>fabs(7.5) | 7.5<br>7.5 | cmath |
| ceil | Ceiling (round up) | double | double | ceil(3.2)<br>ceil(3.9) | 4.0<br>4.0 | cmath |
| floor | Floor (round down) | double | double | floor(3.2)<br>floor(3.9) | 3.0<br>3.0 | cmath |
| exit | End program | int | void | exit(1); | None | cstdlib |
| rand | Random number | None | int | rand( ) | Varies | cstdlib |
| srand | Set seed for rand | unsigned int | void | srand(42); | None | cstdlib |

# Functions

## The `main()` Function

"Special" function, serves as entry point to the program.
Only one `main()` can exist in a program.
Called by the Operating System, not by the programmer!
Should `return` an integer (`0` is traditional, Clean-termination/No-error return code).

## Function Functionalities

- Build "blocks" of programs
- Divide and conquer large problems
- Increases readability and reusability
- Separate source files from main() for easy sharing.

**Note:**
Functions in **C++** can only `return` one thing!
(one type of variable)

- This might seem limited, for now…

## Functions & Parameters

Methods of passing arguments to functions:

- Pass-by-Value:
  A "*Copy*" of the value of the actual argument is used.

- Pass-by-Reference:
  The "*Actual*" argument itself is used.

- Pass-by-Address:
  A "*Copy*" of the value of the argument is used …

(*but:*) the argument is a special type that allows to in-directly use another variable.

## Pass-by-Value

A simple function that adds `1` to an integer and `return`s the new value:

Declaration:

```
int addOne (int  num) {
    return ++num;
}
```

Call:

```
int enrolled = 99;
addOne(enrolled);
```

`enrolled: 99`

When the `addOne()` is called, the *value* of the variable is passed in as an argument. The *value* is saved in `addOne()`'s *local variable* `num`.

- Remember Variable Scope!
  Changes made to *local variables* do not affect anything outside of Scope Block.
  The `main()` and `addOne()` can't see each other's variables.

## Pass-by-Value

A simple function that adds `1` to an integer and `return`s the new value:

Declaration:

```
int addOne (int  num) {
    return ++num;
}
```

Call:

```
int enrolled = 99;
addOne(enrolled);                    enrolled: 99
enrolled = addOne(enrolled);         enrolled: 100
```

*Copy* of actual argument passed.

- Considered "local variable" inside function.
- If modified, only "local copy" changes.

Function has no access to *Actual* argument from caller.

- This is the "default" method.

## Pass-by-Value

A common mistake:

- Declaring parameter "again" inside the function:

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;
    int minutesWorked
}
```

Local Variable

Shadowing

Compiler error: `"error: declaration of 'int minutesWorked' shadows a parameter…"`

- Parameters are like *local* variables.
- Function will "declare and create them" automatically.

## Pass-by-Reference ( & )

Provides access to caller's *Actual* argument.

- Caller's data *can* be modified by called function!

Typically used for input function.

- To retrieve data for caller.

Specified by ampersand (&) after type in formal parameter list (Function Definition).

```
<return type> <function_name> (<parameter type> & );
```

```
void squareThisNumber (int & n);
```

by-Reference: *Actual* Argument

*vs*

```
int squareNumber (int n);
```

by-Value: *Copy-of* Argument

## Pass-by-Reference ( & )

What is really passed in:

- Reference to something that exists outside of the function scope.
  The "*Actual*" argument (*vs* its "*Copy*").

*Example* (with contrast to Pass-by-Value model) :

Declaration:

```
void addOne (int & num) {
    ++num;
    return;
}
```

Call:

```
int enrolled = 99;
addOne(enrolled);
```

enrolled: 100

## Pass-by-Reference ( & )

```cpp
//Program to demonstrate call-by-reference parameters.
#include <iostream>
using namespace std;

void getNumbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swapValues(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void showResults(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}
```

No need to **`return`** anything!

```cpp
void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

## `const`-Reference Parameters (`const &`)

Calling-by-Reference arguments is inherently "dangerous":

- Caller's data can be changed, sometimes NOT desirable behaviour.

Common technique to "protect" data:

The keyword `const`.

```
void sendConstRef(const int & par1, const int & par2);
```

- No changes allowed inside function body, arguments have "read-only" qualification.

Note: `const`-Reference yields different *Function Signature* than non-`const` Reference…

These are 2 separate functions and each can be used to do a different thing (separate implementions).

```
meters.feet(const double & ft);
meters.feet(double & ft);
```

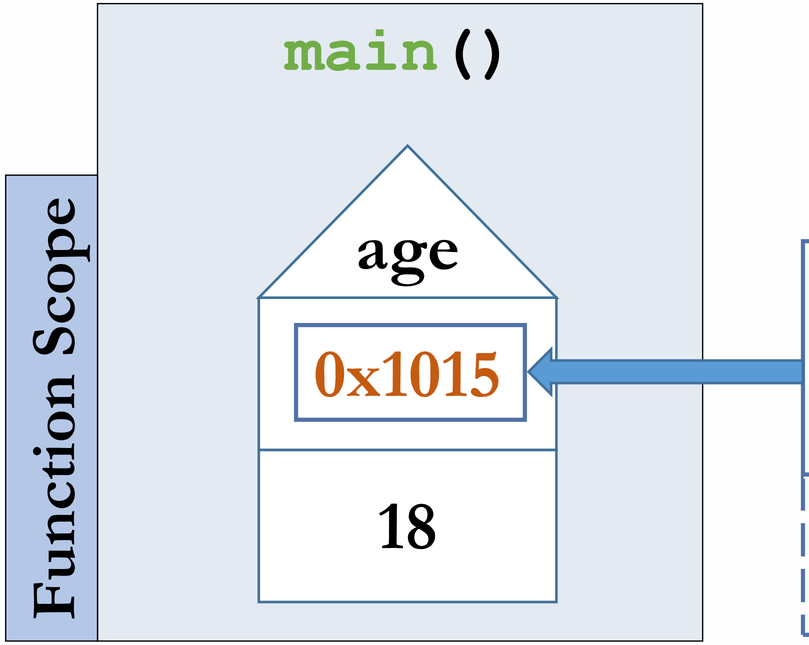
# Pointers

## Addresses

Remember the Pass-by-Value model :

```
int age = 18;
age = addOne( age );
```

```
int addOne ( int  num ) {
        return ++num;
}
```




This is the variable Address:
Addresses commonly represented
by HEX numbers

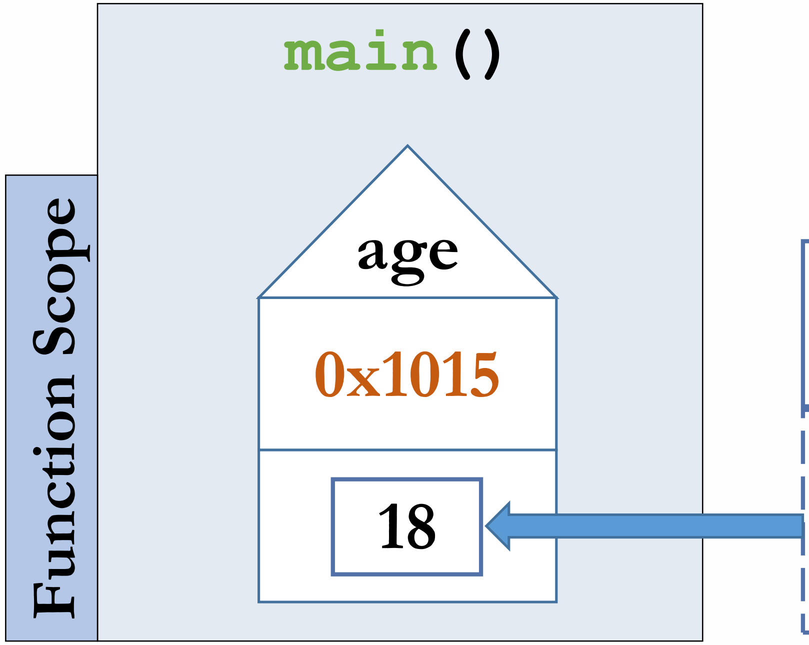
Where it "lives" in the program memory,
the starting memory location

## Addresses

Remember the Pass-by-Value model :

```
int age = 18;
age = addOne( age );
```

```
int addOne ( int  num ) {
        return ++num;
}
```




This is the variable Value:
Its value semantics are determined by its type.

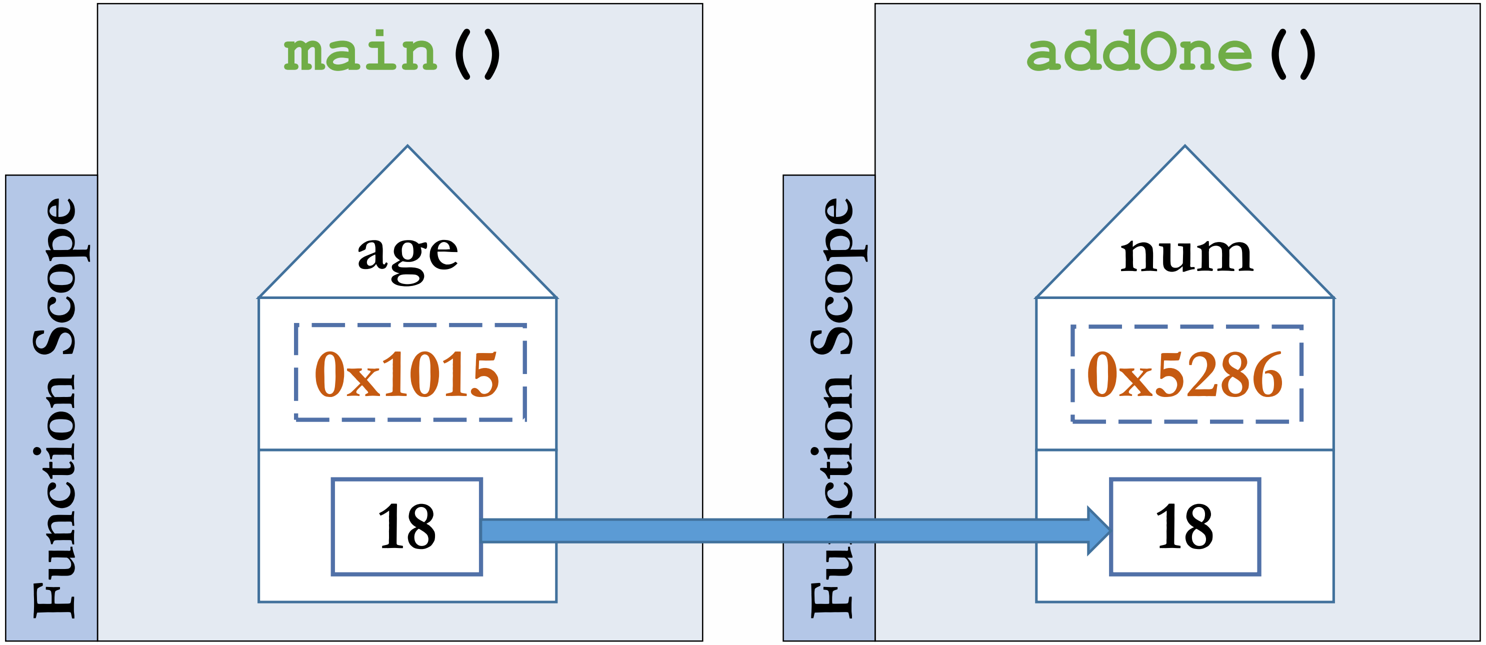
It is a representation of the memory content (sequence of bytes) starting at its Address and with a total number defined by the type.

## Addresses

Remember the Pass-by-Value model :

```
int age = 18;
age = addOne( age );
```

```
int addOne ( int  num ) {
        return ++num;
}
```



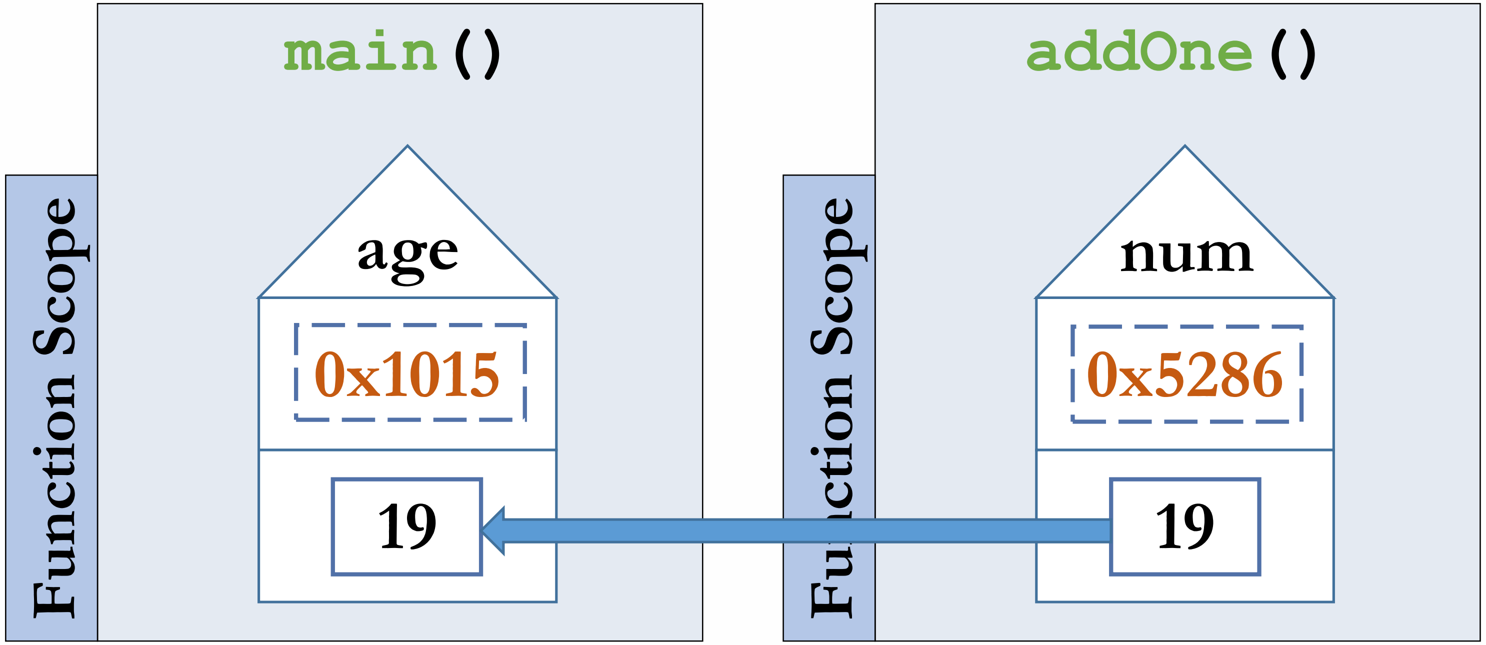

Addresses commonly HEX numbers

## Addresses

Update via **return** value and assignment:

```
int age = 18;
age = addOne( age );
```

```
int addOne ( int  num ) {
        return ++num;
}
```

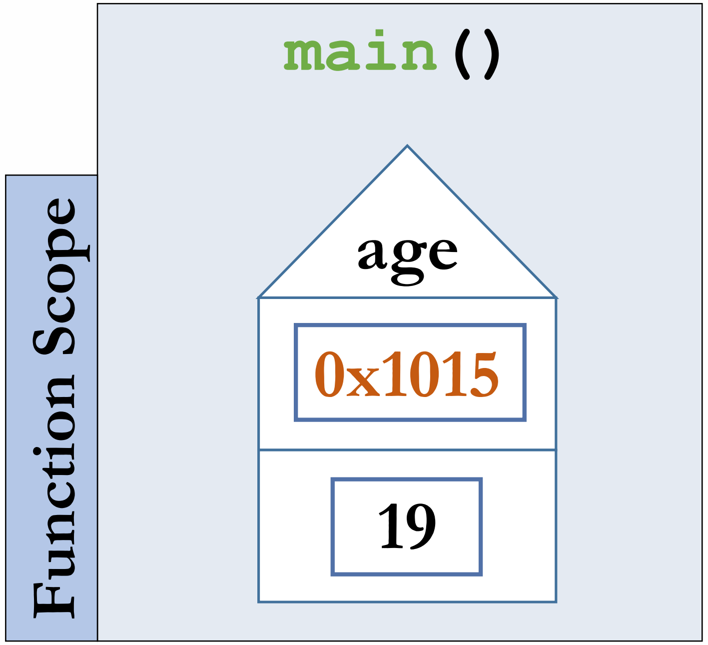



**age**, **num**
in separate
Scopes.

## Addresses

After Function Call:

```
int age = 18;
age = addOne( age );
```




```
int addOne ( int  num ) {
        return ++num;
}
```

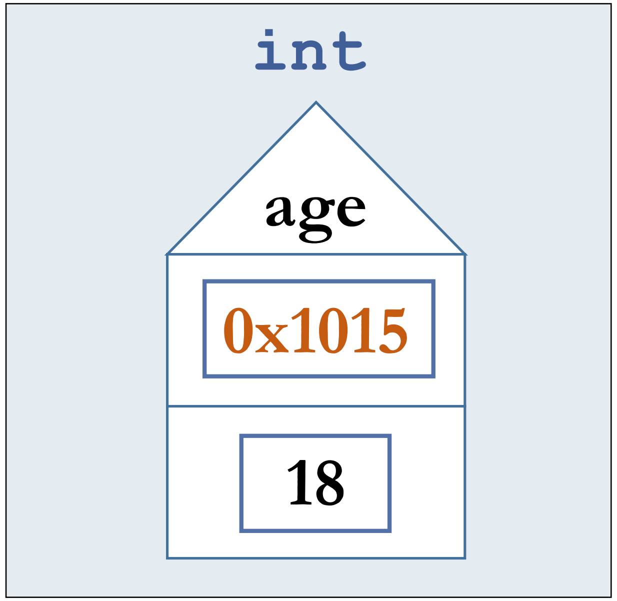
**num** not reachable outside this Block Scope (would not be available even if it were a **static** variable, it is not a life-time issue)

# Pointers

## "Addresses-of" Operator ( & )

To get the *Address-Of* a variable we pre-pend the ampersand (&) operator to its name.

```
int age = 18;
```




```
cout << age;     Output: 18

cout << &age;    Output: 0x1015
```

## Pointer

A *Variable* whose Value holds the *Address-Of* something somewhere in memory.

```
int x = 37;
int * ptr = (int *)0x7ffedcaba5c4;
cout << "x   is " << x << endl;
cout << "ptr is " << ptr << endl;
```

Typecasting a `long` number to an `int *` value here to set a specific HEX value to the pointer.
(This is just for demonstration in this context and is never done in practice, because otherwise the value of the pointer variable would be left uninitialized !)

This will print out something like:

```
x   is 37
ptr is 0x7ffedcaba5c4
```

Addresses commonly HEX numbers

## Pointer Utility

Pointers are incredibly useful in programming.

- Allow functions to:
  Modify multiple arguments.
  Use and modify arrays as arguments.

- Increase program (compiled function) efficiency.

- Creation / handling / use of Dynamic Objects (more on that later).

## Pointer Declaration

A pointer is just like any regular variable. It has:

- Type
- Name
- Value (what kind?)

Pointer declaration /creation requires the (*) symbol.

```
int x = 37;
int * ptr = NULL;

cout << "x   is " << x << endl << "ptr is " << ptr << endl;
```

*Note*: Typical pointer initialization to value **NULL** ensures that we don't end up with a random uninitialized value !

## Pointer Declaration

Valid pointers declaration / creation statements:

```
int *ptr1;
int* ptr2;
int * ptr3;
int*ptr4;
```

Just avoid the last one.

Note:

- Multiple pointers inline declaration / creation:

```
int * ptr1, * ptr2, * ptr3;
```

```
int * ptr1, ptr2, ptr3;
```

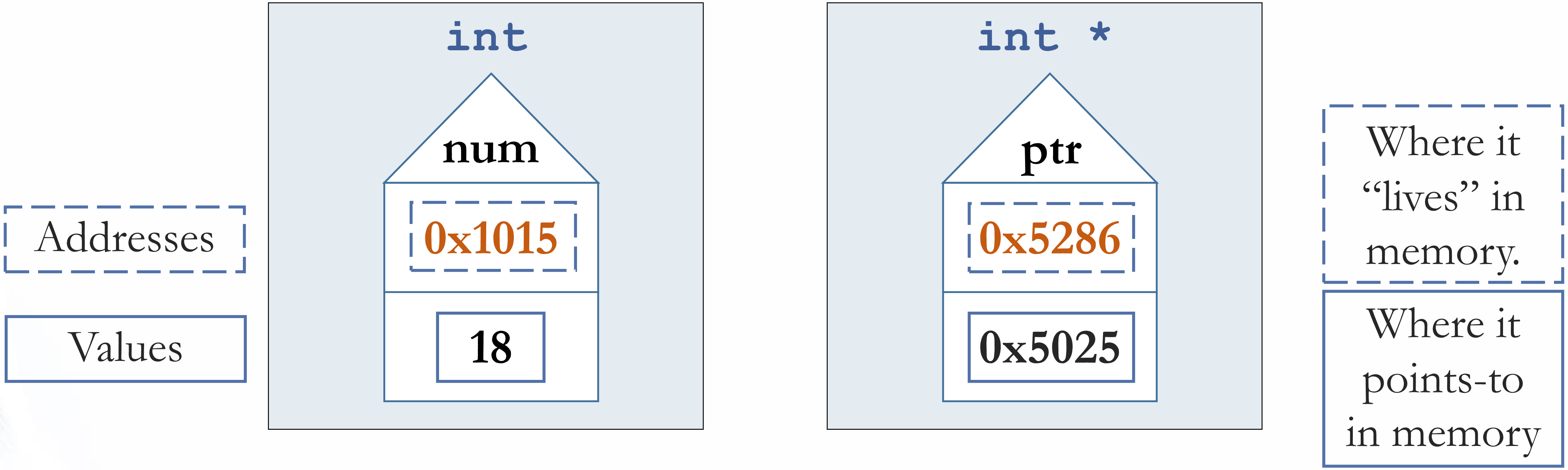
Looks right, but no,
this declares an `int *` and 2 `int`s

# Pointers

## Pointer Value

As earlier stated, pointers are "Just Variables".

➢ Pointer's Value: an *Address* in memory (instead of storing an `int`/`float`/`char`/etc.)
  Note: Pointer's size in memory is not guaranteed (implementation-defined).



## Pointer Assignment

Value (pointed-to Address) assignment:

- To get the *Address-Of* a variable we use the ampersand (**&**) operator.

```
int  x = 5;
int * xPtr = NULL;
xPtr = &x;
```

Simple grammar:
"Pointer-value gets assigned the Address-of variable **x**".

- Pointer-to-pointer assignment (also valid):

```
int * yPtr;
yPtr = xPtr;
```

## Pointer Assignment

Value (pointed-to Address) assignment:

*Address-Of* a Value is not enough for a valid assignment!

- Pointer has a Type.

```
int   x = 5;
char * ptr5 = &x ;
```

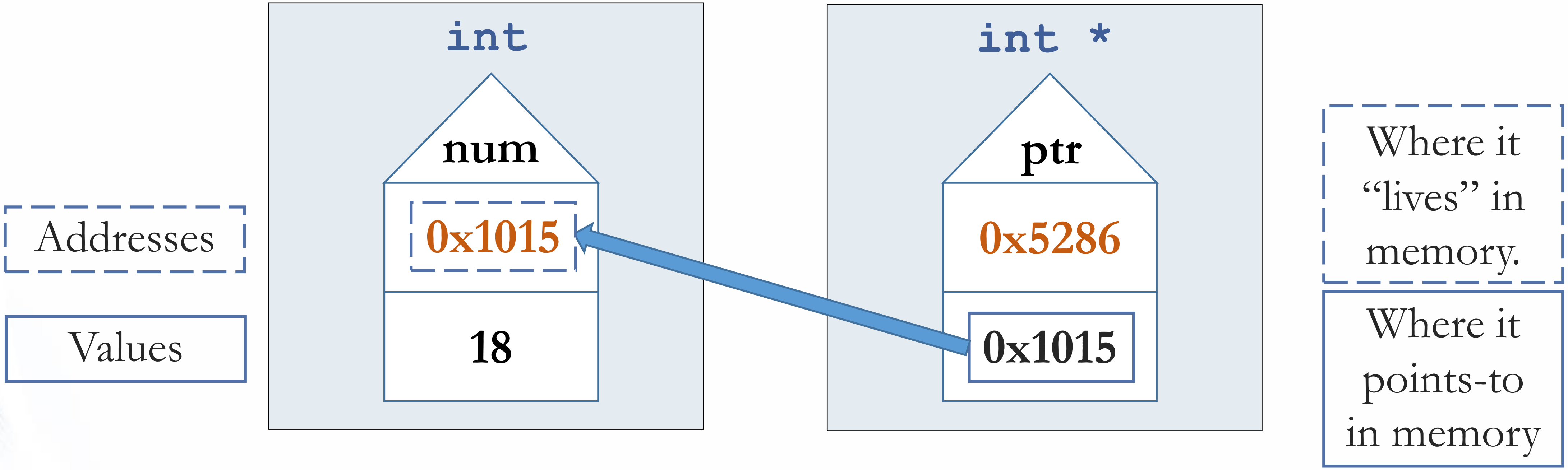
Pointer type must match the type of the variable whose address it stores.

Compiler error: "`error: cannot convert 'int*' to 'char*' in initialization.`"

## Pointer Assignment

Assignment means telling the pointer what memory address to point to:

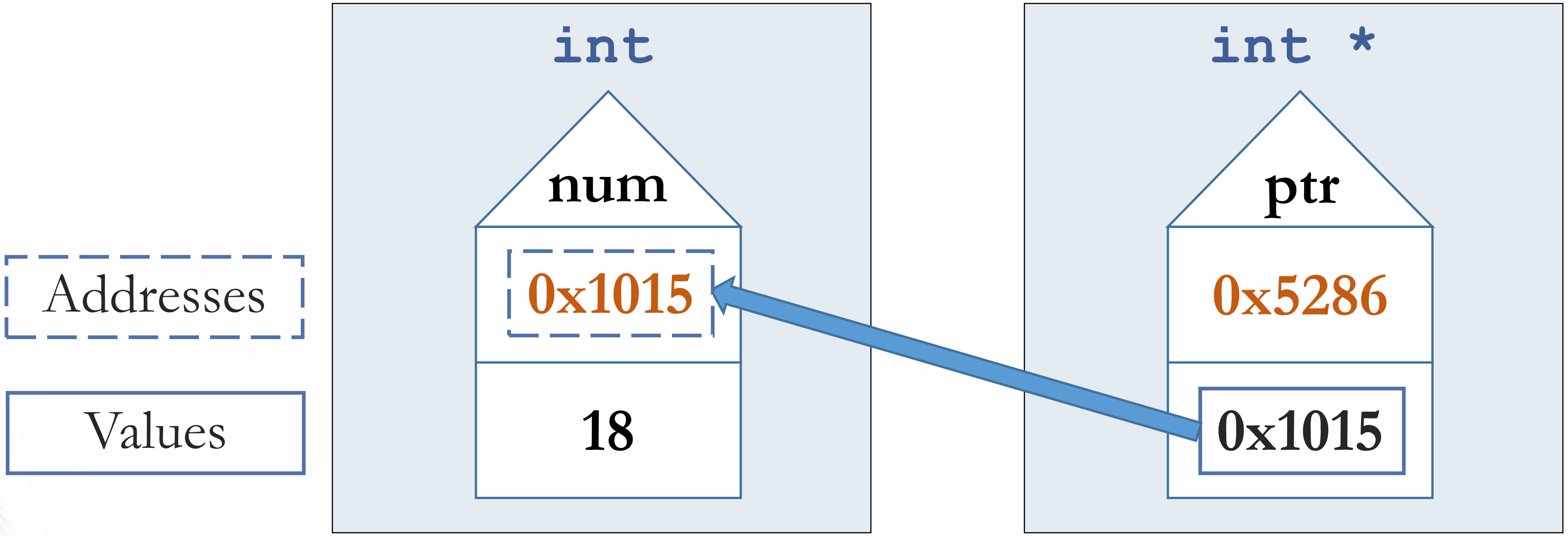
```
int num = 18;
int * ptr = &num;
```



**Indirection** (Dereference) **Operator** ( * ) or "Value-Pointed-By"

To refer to the *Value-Pointed-By* a pointer, we pre-pend the star (*) operator to its name.

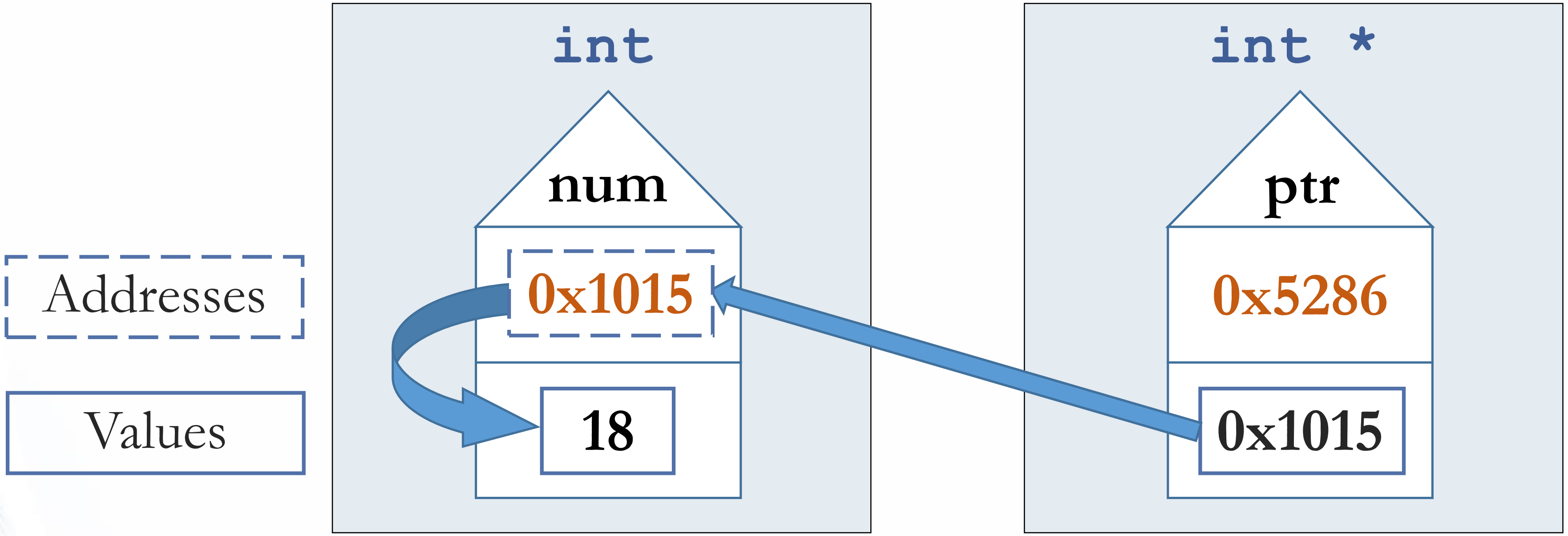
```
… = *ptr
*ptr = …;
```



**Indirection (Dereference) Operator** ( * ) or "Value-Pointed-By"

To refer to the *Value-Pointed-By* a pointer, we pre-pend the star (*) operator to its name.

```
… = *ptr
*ptr = …;
```



**Indirection (Dereference) Operator** ( `*` ) or “Value-Pointed-By”

At this point what follows depends on purpose of Dereferencing.

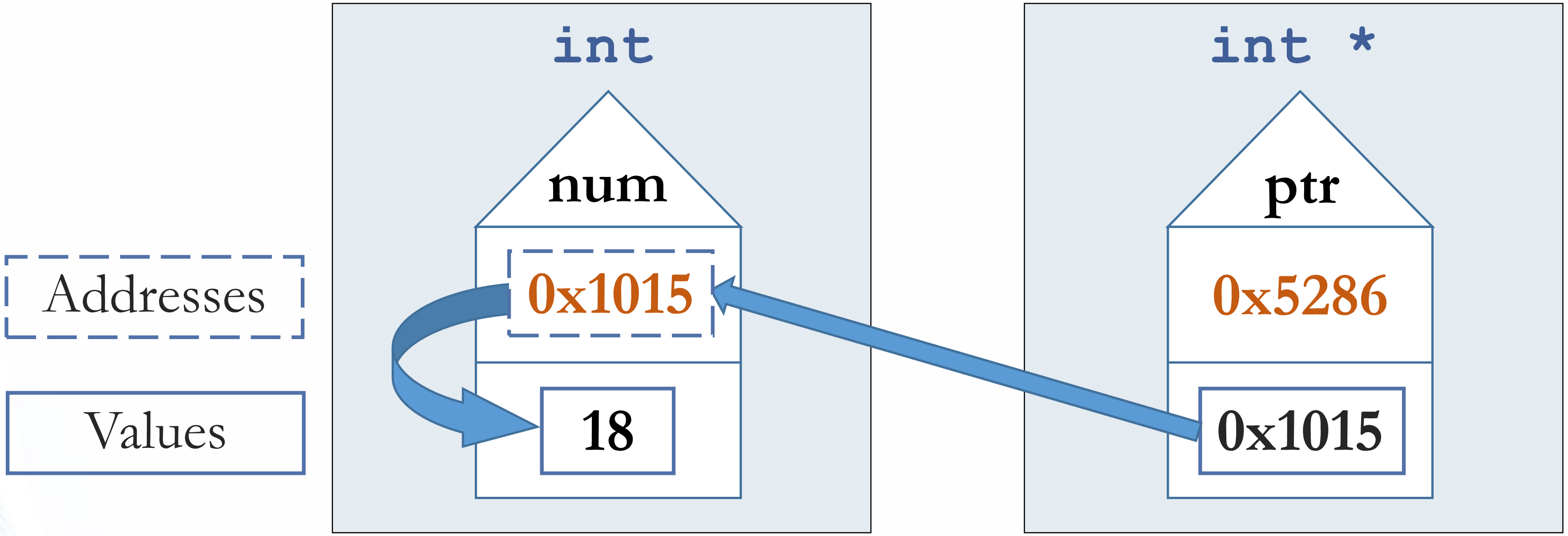
A Dereference can be in three “places”:

- On the *left hand* side of the assignment operator.
- On the *right hand* side of the assignment operator.
- In an expression with *no assignment* operator (e.g. a `cout` statement).

**Indirection (Dereference) Operator** ( * ) or "Value-Pointed-By"

To refer to the *Value-Pointed-By* a pointer, we pre-pend the star (*) operator to its name.

```
int inVar = *ptr;
cout << *ptr << endl;
```

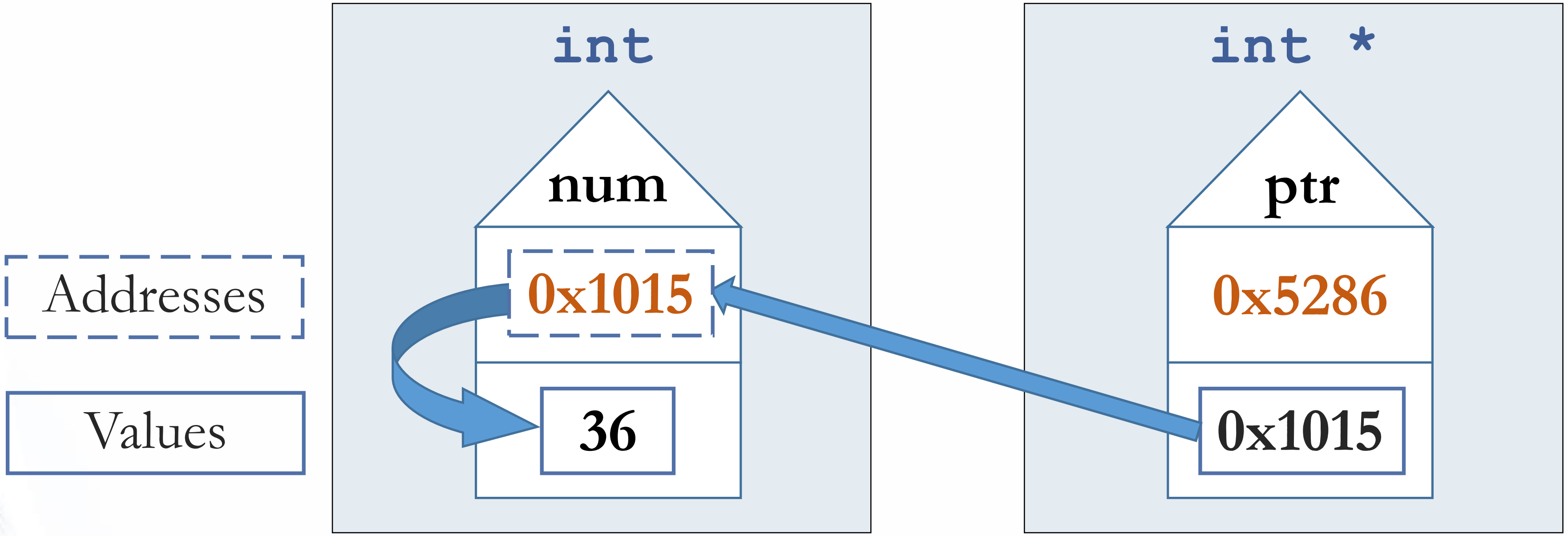



Access variable.
*Get* its value.

**Indirection (Dereference) Operator** ( * ) or "Value-Pointed-By"

To refer to the *Value-Pointed-By* a pointer, we pre-pend the star (*) operator to its name.

```
*ptr = 36;
```




Access variable.
*Change* its value.

## Pointers as Function Parameters

Common Paradigm:

- A function that modifies more than one values.
  Example: How to multiply Two `int` values by an order of magnitude.

```
void increaseOrder( <two ints> ) {
    // multiply first int by 10
    // multiply second int by 10
    // have the values persist after control is return'ed -- how?
}
```

- Can't use Pass-by-Value, then return & assign method.
  `return` will only give back One value.
- Can use Pass-by-Reference (working directly on passed arguments).
- But also …

## Pointers as Function Parameters

Common Paradigm:

➢ A function that modifies more than one values.
  Example: How to multiply Two `int` values by an order of magnitude.

Work on an Address basis.

```
void increaseOrder( int * ptr1, int * ptr2) {
    // multiply by ten the values of the ints that ptr1, ptr2 point to
    *ptr1 = *ptr1 * 10;
    *ptr2 *= 10;
    // return nothing
}
```

## Pointer Parameters in Functions

Function Declaration:

```
void increaseOrder(int * ptr1, int * ptr2);
```

Function Call:

```
   int firstNum = 25;
   int secondNum = 350;
   int * firstNumPtr = &firstNum;
   int * secondNumPtr = &secondNum;
   increaseOrder(firstNumPtr, secondNumPtr);
or increaseOrder(&firstNum, &secondNum);
or increaseOrder(firstNumPtr, &secondNum);
```

Note:

```
increaseOrder(&25, &350);
```

Note: Won't work, these are Literals!

"error: lvalue required as unary '&' operand"

## Pointers at Work

`int` variable instantiation – Memory allocation

```
int  x = 5;
```

| | |
|---|---|
| Variable name | **x** |
| Memory Address | 0x7f96c |
| Value | 5 |

## Pointers at Work

`int` Pointer variable instantiation – Value assignment by Reference (*Address-Of*)

```
int  x = 5;
int * xPtr = &x;  /* xPtr points to x */
```

| Variable name | x | xPtr |
|---|---|---|
| Memory Address | 0x7f96c | 0x7f960 |
| Value | 5 | 0x7f96c |

## Pointers at Work

`int` variable instantiation – Value assignment by Dereferencing (*Value-Pointed-By*)

```
int  x = 5;
int * xPtr = &x;   /* xPtr points to x */
int  y = *xPtr;    /* y's value is now ... */
```

| Variable name | x | xPtr | y |
|---|---|---|---|
| Memory Address | 0x7f96c | 0x7f960 | $\boxed{\texttt{0x7f95c}}$ |
| Value | 5 | 0x7f96c | |

## Pointers at Work

`int` variable instantiation – Value assignment by Dereferencing (*Value-Pointed-By*)

```
int  x = 5;
int * xPtr = &x;   /* xPtr points to x */
int  y = *xPtr;    /* y's value is now ... */
```

| Variable name | x | xPtr | y |
|---|---|---|---|
| Memory Address | 0x7f96c | 0x7f960 | 0x7f95c |
| Value | 5 | 0x7f96c | |

## Pointers at Work

`int` variable instantiation – Value assignment by Dereferencing (*Value-Pointed-By*)

```
int   x = 5;
int * xPtr = &x;   /* xPtr points to x */
int   y = *xPtr;   /* y's value is now ... */
```

| Variable name | x | xPtr | y |
|---|---|---|---|
| Memory Address | 0x7f96c | 0x7f960 | 0x7f95c |
| Value | 5 | 0x7f96c | |

## Pointers at Work

`int` variable instantiation – Value assignment by Dereferencing (*Value-Pointed-By*)

```
int  x = 5;
int * xPtr = &x;   /* xPtr points to x */
int  y = *xPtr;    /* y's value is now 5 */
```

| Variable name | x | xPtr | y |
|---|---|---|---|
| Memory Address | 0x7f96c | 0x7f960 | 0x7f95c |
| Value | 5 | 0x7f96c | 5 |

## Pointers at Work

`int` variable Value assignment – No address aliasing / No variable correlation

```
int  x = 5;
int * xPtr = &x;   /* xPtr points to x */
int  y = *xPtr;    /* y's value is now 5 */
x = 3;             /* y is still 5 */
```

| Variable name | x | xPtr | y |
|---|---|---|---|
| Memory Address | 0x7f96c | 0x7f960 | 0x7f95c |
| Value | 3 | 0x7f96c | 5 |

## Pointers at Work

`int` variable Value assignment – No address aliasing / No variable correlation

```
int  x = 5;
int * xPtr = &x;   /* xPtr points to x */
int  y = *xPtr;    /* y's value is now 5 */
x = 3;             /* y is still 5 */
y = 2;             /* x is still 3 */
```

| Variable name | x | xPtr | y |
|---|---|---|---|
| Memory Address | 0x7f96c | 0x7f960 | 0x7f95c |
| Value | 3 | 0x7f96c | 2 |

## Reference-Types

Reference-Type variable declaration with the ampersand (**&**) symbol.

```
int x = 10;
int & xRef = x;
```

Once created, they don't need the ampersand (**&**) or asterisk (*****) in their use.

- They are actually "*Aliases*" to pre-existing variables.
  (They look like normal variables)

Rules:

- References *must* be initialized at declaration (they have to *Alias* something).
  Once initialized, they are forever tied to the thing they reference.
  No such thing as a **NULL** *reference* (unlike a **NULL** *pointer*).
- References cannot be changed (any attempt to assign just references the aliased variable).
- References are another "name" for a variable (dereferencing does not make sense).

## Reference-Types

Reference-Type variable declaration with the ampersand (**&**) symbol.

```
int x = 10;
int & xRef = x;
```

Once created, they don't need the ampersand (**&**) or asterisk (*****) in their use.

- They are actually "*Aliases*" to pre-existing variables.
  (They look like normal variables)

Rules:

- From the C++11 standard:

**[dcl.ref]** [...] a **NULL** reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a **NULL** pointer, which causes undefined behavior.

## Reference Caveats

Reference-Type variable declaration with the ampersand (&) symbol.

```
int & xRef = x;
```

- Essentially Pass-by-Reference is achieved by passing Reference-Type parameters.
- Using them looks identical to using a value (Is easier always a good thing?) May easily think you're passing by value…

```
void changeByRef (int x){
   x = x + 1;
   cout << "changeByRef " << x << "\n";
}
```

```
int x = 1;
changeByRef( x );
cout << "main " << x << "\n";
```

Output:
```
changeByRef 2
main 1
```

## Reference Caveats

Reference declaration the ampersand (&) symbol.

```
int & xRef = x;
```

- Essentially Pass-by-Reference is achieved by declaring Reference-type parameters.
- Using them looks identical to using a value (Is easier always a good thing?) May easily think you're passing by value…

```
void changeByRef (int & x){
   x = x + 1;
   cout << "changeByRef " << x << "\n";
}
```

```
int x = 1;
changeByRef( x );

cout << "main " << x << "\n";
```

Output:
```
changeByRef 2
main 2
```

# Overview: Parameters in Functions

## i) Pass-by-Value

- The "default" way.
- Implies *Data Copy* operation.

```
void printVal (int x);

int x = 5;
int * xPtr = &x;

printVal(x);
printVal(*xPtr);
```

Valid Calls

# Overview: Parameters in Functions

## ii) Pass-by-Address

- Uses pointers, and uses (*) and (&) operators.
- *Address* passed (via Pointer value), *Data Copy* unnecessary.

```
void changeVal (int * x);

int x = 5;
int * xPtr = &x;

changeVal(&x);
changeVal(xPtr);
```

Valid Calls

# Overview: Parameters in Functions

## ii) Pass-by-Address

- Uses pointers, and uses (*) and (&) operators.
- *Address* passed (via Pointer value), *Data Copy* unnecessary.

```
void changeVal (int * x);

int x = 5;
int * xPtr = &x;

changeVal(&x);
changeVal(xPtr);
```

No guarantees pointer is valid.

Valid Calls

Note:
Have to check for **NULL** pointer inside function calls !

## iii) Pass-by-Reference

- Uses (&) operator once (function declaration).
- *Actual* Argument passed, *Data Copy* unnecessary.

```
void changeByRef (int & x);

int x = 1;
int & xAlias = x;

changeByRef(x);
changeByRef(xAlias);
```

Valid Calls

# Overview: Parameters in Functions

## iii) Pass-by-Reference

- Uses (**&**) operator once (function declaration).
- *Actual* Argument passed, *Data Copy* unnecessary.

```
void changeByRef (int & x);

int x = 1;
int & xAlias = x;

changeByRef(x);
changeByRef(xAlias);
```

Valid Calls

Note:
Variable might be changed.
Have to bear in mind the function prototype !

**Pass-by-Address**

Arrays "*Decay*" into Pointers, they are always Passed-by-Address to functions.

- Program does not make a copy of an array.
- Changes made to an array inside a function will persist after the function exits.

Remember entire Arrays as Function Arguments:

```
double array[10] = {};  // braced initializer for zero-initialization
```

or

```
void arrayWholeFunction(double vals [], int num);
void arrayWholeFunction(double * vals , int num);
```

Valid Definitions

or

```
arrayWholeFunction(array, 10);
arrayWholeFunction(&array[0], 10);
```

by-Address (name)
by-Address-of (1st element)

# *Note*: Functions and C-strings

## Pass-by-Address

C-strings are `char` type arrays, they are always Passed-by-Address to functions.

- Same as any other array.

Remember entire Arrays as Function Arguments (nothing more special):

```
char mystring[] = "Hello world!";  // string literal for initialization
```

or

```
void capitalizeFirstLetter(char text []);
void capitalizeFirstLetter(char * text );
```

Valid Definitions

or

```
capitalizeFirstLetter(mystring);
capitalizeFirstLetter(&mystring[0]);
```

by-Address (name)
by-Address-of (1st element)

# Pointer Arithmetic

## Moving through an Array with Pointers

```
int num_array[10] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 };
int* num_ptr;
num_ptr = num_array; //or equivalently, num_ptr = &num_array[0];
cout  << *num_ptr;
```

0 Pointer points to Address of array 1st element.

## Moving through an Array with Pointers

```
int num_array[10] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 };
int* num_ptr;
num_ptr = num_array; //or equivalently, num_ptr = &num_array[0];
cout  << *num_ptr;
num_ptr++;
cout  << *num_ptr;
```

**1** Pointer moves to point 1 position ahead (++) in memory, therefore pointing to Address of array $2^{nd}$ element.

# Pointer Arithmetic

## Moving through an Array with Pointers

```
int num_array[10] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 };
int* num_ptr;
num_ptr = num_array; //or equivalently, num_ptr = &num_array[0];
cout  << *num_ptr;
num_ptr++;
cout  << *num_ptr;
num_ptr += 8;
cout  << *num_ptr;
```

9 Pointer moves to point 8 positions more ahead (+=8) in memory, therefore pointing to Address of array 10th element.

# Pointer Arithmetic

## Moving through an Array with Pointers

```
int num_array[10] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 };
int* num_ptr;
num_ptr = num_array; //or equivalently, num_ptr = &num_array[0];
cout  << *num_ptr;
num_ptr++;
cout  << *num_ptr;
num_ptr += 8;
cout  << *num_ptr;
num_ptr--;
cout  << *num_ptr;
```

8 Pointer moves to point 1 position backwards (--) in memory, therefore pointing to Address of array 9th element.

## Moving through an Array with Pointers

```
int num_array[10] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 };
int* num_ptr;
num_ptr = num_array; //or equivalently, num_ptr = &num_array[0];
cout  << *num_ptr;
num_ptr++;
cout  << *num_ptr;
num_ptr += 8;
cout  << *num_ptr;
num_ptr--;
cout  << *num_ptr;
num_ptr-=5;
cout  << *num_ptr;
num_ptr = num_array;
cout  << *num_ptr;
```

3 Pointer moves to point 4 positions more back (-=5) in memory, therefore pointing to Address of array 4th element.

## Moving through an Array with Pointers

```
int num_array[10] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 };
int* num_ptr;
num_ptr = num_array; //or equivalently, num_ptr = &num_array[0];
cout  << *num_ptr;
num_ptr++;
cout  << *num_ptr;
num_ptr += 8;
cout  << *num_ptr;
num_ptr--;
cout  << *num_ptr;
num_ptr-=5;
cout  << *num_ptr;
num_ptr = num_array;
cout  << *num_ptr;
```

0 Pointer reassigned to points again to Address of array 1st element.

# Comprehensive Pointer Example

```
#include <iostream>
using namespace std;

const int MAX_STR_SIZE = 255;

void cStringPrint(char * cstr);

int main(){
  char my_cString[MAX_STR_SIZE] = "Hello World!";
  cStringPrint ( my_cString );
  return 0;
}
```

A C-string i.e. **char** array

A Function with a **char*** parameter

```
void cStringPrint(char * cstr){
  while( *cstr ){
    cout << *cstr++ ;
  }
}
```

Equivalent to: `cout << *cstr ;` `cstr++;`

A Function that:

- Iterates through an array by employing Pointer-Arithmetic (**++**)
- Accesses elements of an array by employing Pointer-Dereferencing (*****)

CS-202

Time for Questions !