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 (9:00-12:50)	
	CLASS		CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Your 1st Project Deadline is this Wednesday 9/6.

- ➢ PASS Sessions held Monday-Tuesday, get all the help you may 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

Functions

Function Parts

Function Prototype (Declaration)

- Information for compiler to properly interpret calls.

Function Implementation (Definition)

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

Function Call

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

Function Prototype

Gives compiler information about the function

- How to interpret calls to the function.

```
<return type> <function_name> (<parameters>);
int squareNumber (int n);
```

Semicolon

Must have the parameter's data types.

Placed before any calls

In declaration space of `main()`.

Or above `main()` for global access.

Functions

Function Implementation

Implementation of the function:

```
int squareNumber (int n) {
    int answer = n * n;
    return answer;
}
```

Brackets

- ➢ Function definition must match prototype.
  Placed AFTER the function `main()` (NOT inside).
- ➢ All functions are equal, 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.

Functions

Function Call

Much like a standard C call:

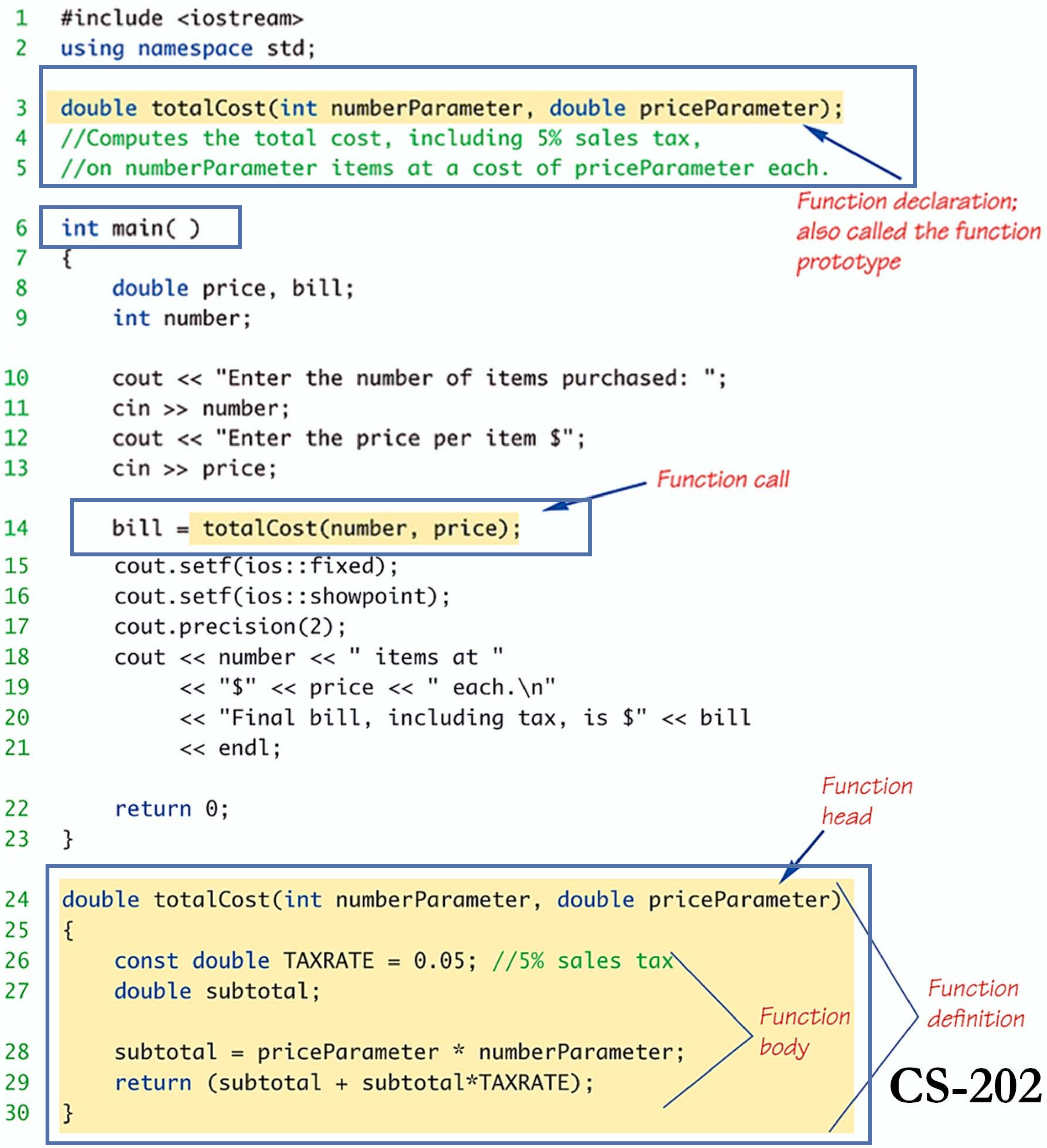
```
    int tenSquared = squareNum(10);
```

- Returns an `int`.
  (Assigned to variable `tenSquared`)
- Arguments: the literal `10`.
  (Can also pass a variable – does it have to be `int`?)

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.

- Multiple return statements?

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

Functions

Function Parameters / Arguments

Variadic Arguments:

```
double precisionMultiSum(int numargs, double arg1, …);
```

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

double precisionMultiSum(int numargs, double arg1 ...){
  va_list ap;
  double sum;
  va_start(ap, arg1);
  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

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 */
```

Functions

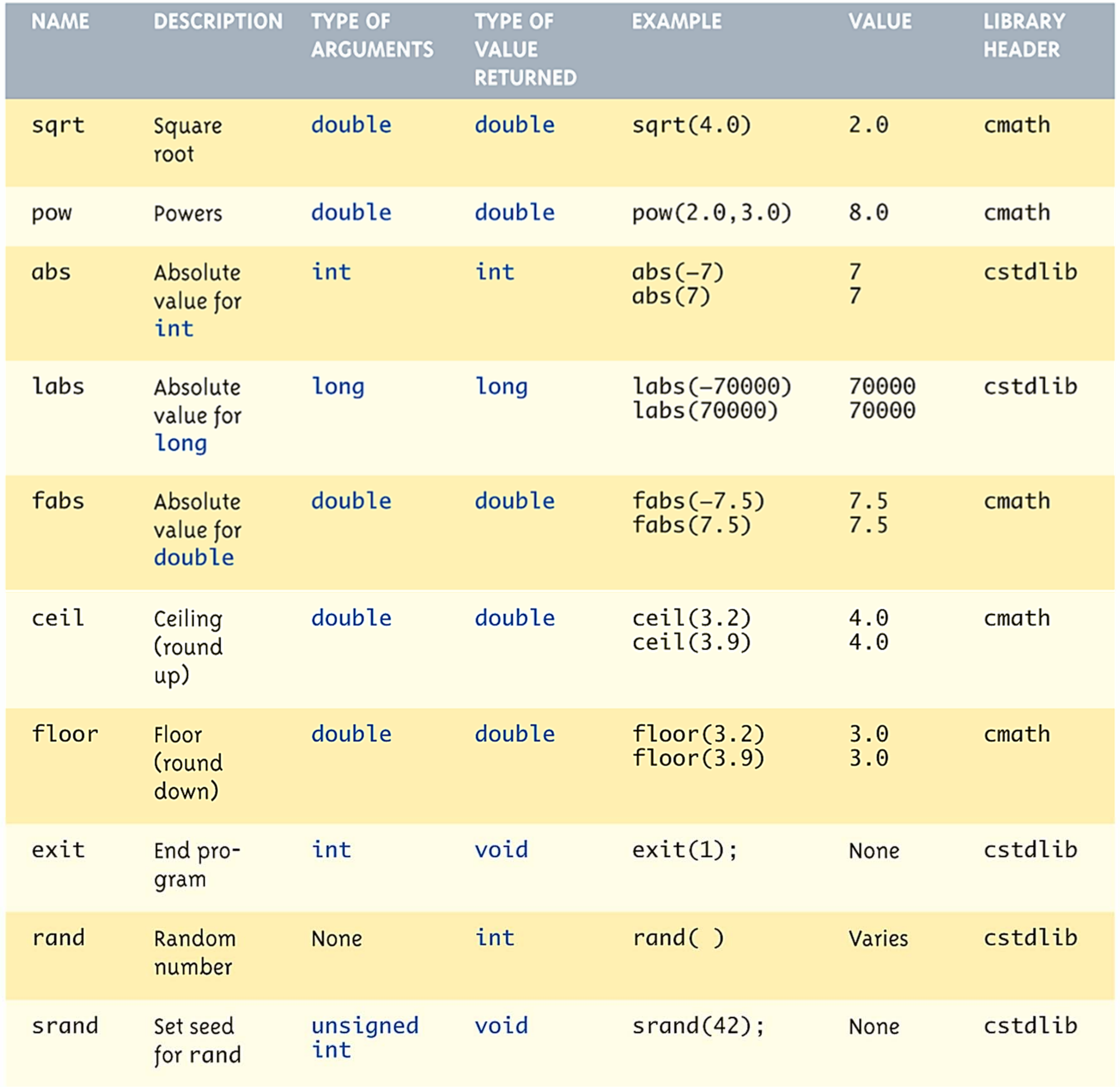
C++ Function Libraries

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

- Original "**C**" libraries:
  <cmath>
  <cstdlib>

- 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)
				abs(7)
					7
					7
						cstdlib

labs	Absolute value for long	long	long	labs(-70000)
				labs(70000)
					70000
					70000
						cstdlib

fabs	Absolute value for double	double	double	fabs(-7.5)
				fabs(7.5)
					7.5
					7.5
						cmath

ceil	Ceiling (round up)	double	double	ceil(3.2)
				ceil(3.9)
					4.0
					4.0
						cmath

floor	Floor (round down)	double	double	floor(3.2)
				floor(3.9)
					3.0
					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 value type)

➢ This might seem limited, for now…

Functions

Functions & Arguments

Two methods of passing arguments as parameters:

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

- Call-by-Reference:
  The "*Address-Of*" the actual argument is used.

Functions

Call-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;
enrolled = AddOne(enrolled);
```

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.

Functions

Call-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;
enrolled = AddOne(enrolled);
```

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.

Call-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…"`

- ➢ Value arguments are like *local* variables.
- ➢ Function will "declare and create them" automatically.

Call-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* Variable

vs

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

by-Value: *Copy-of* Variable

Functions

Call-by-Reference (&)

What is really passed in:

- ➢ Reference to *Memory* location of *Actual* argument.
  The "*Address-Of*" (vs "*Value-Of*"), which is a unique distinct place in memory.

Remember Arrays as Function Arguments:

- ➢ Not the same – We'll see shortly! (But follows the similar rationale)

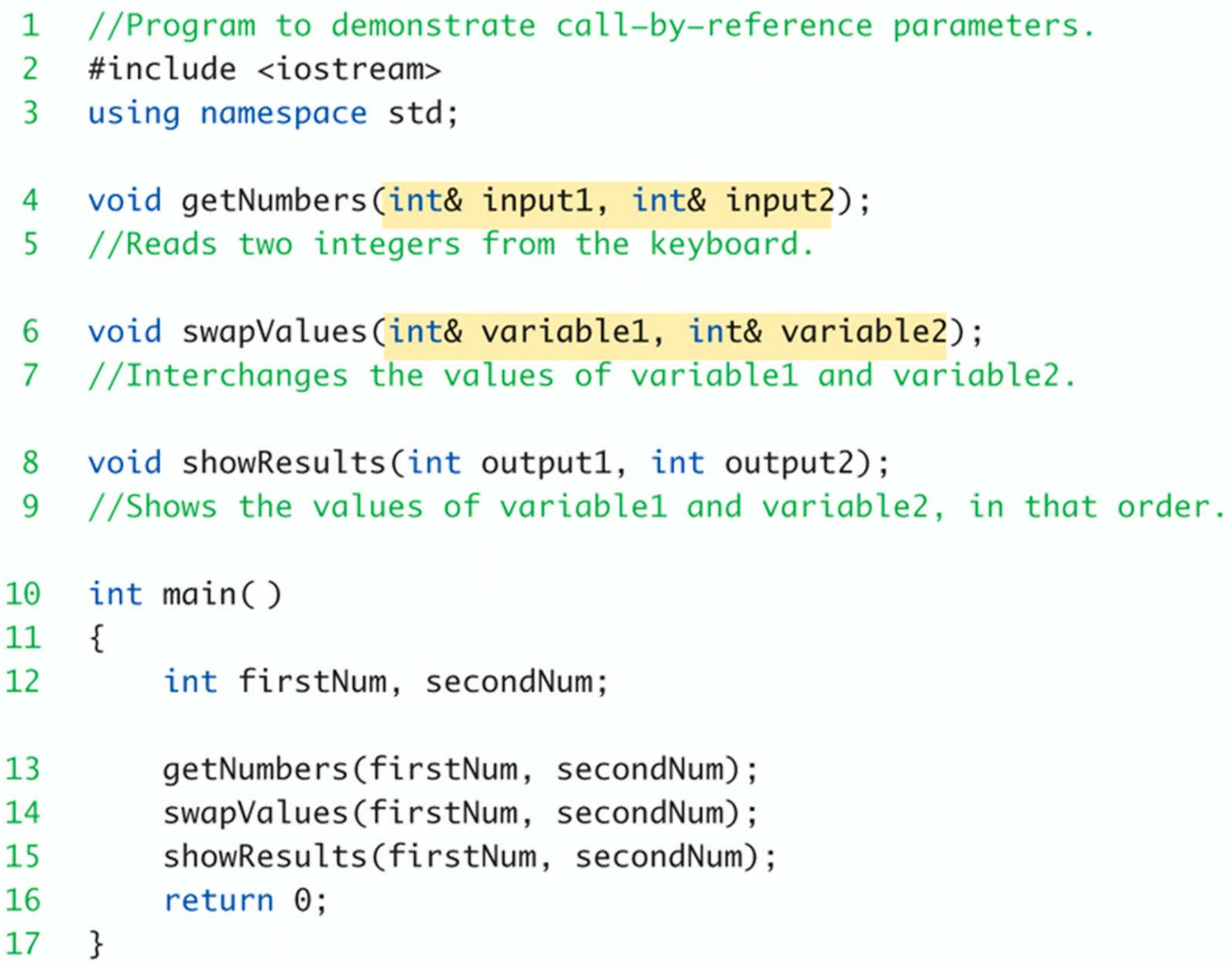
```
double array[10];

void arrayValueFunction(double val);
arrayValueFunction(array[0]);

void arrayWholeFunction(double vals[], int num);
arrayWholeFunction(array, 10);
```

Array Element

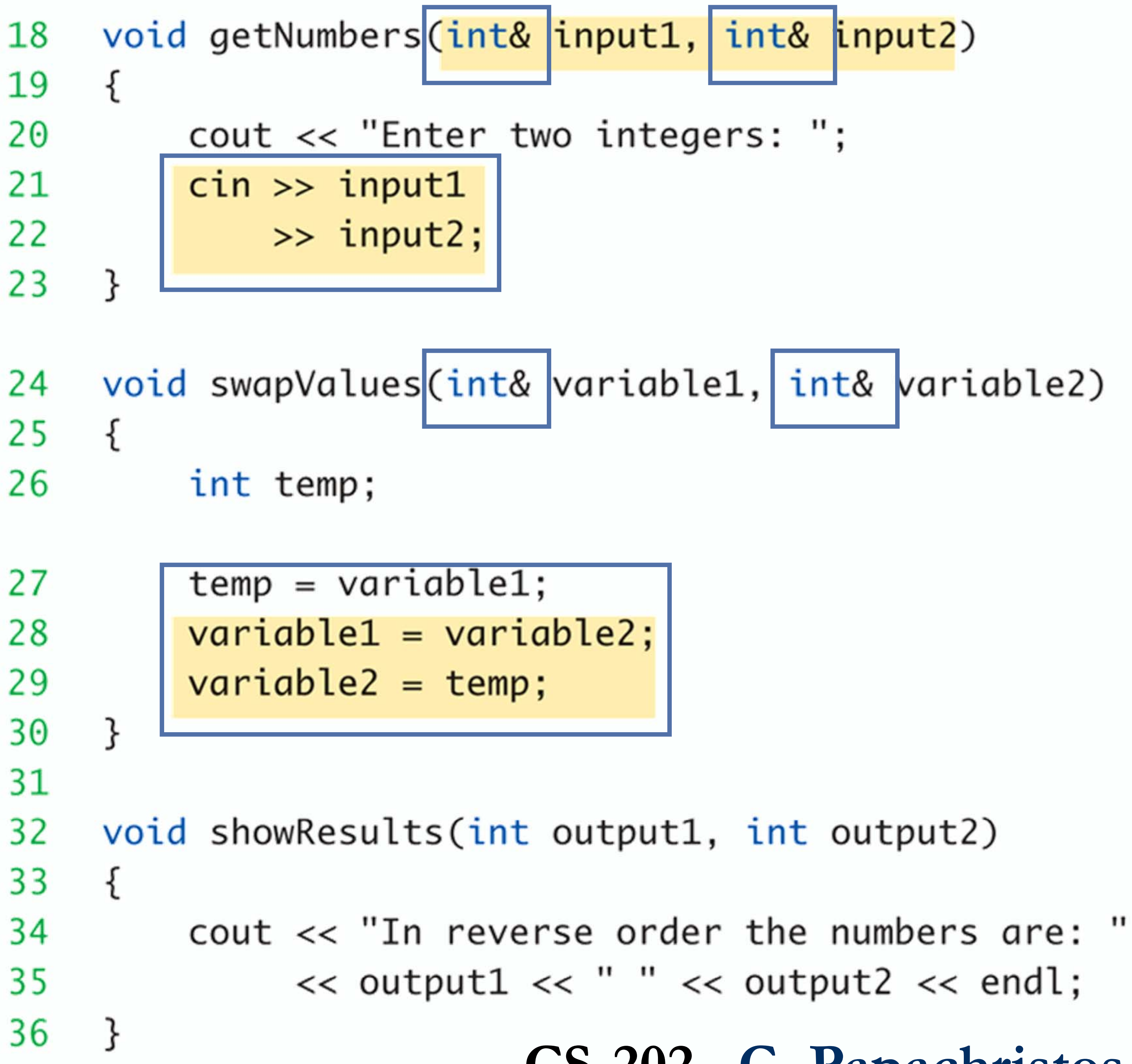
Entire Array
Address-of 1st Element

Functions

Call-by-Reference (&)



```
//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;
}
```



```
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;
}
```

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

Constant-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: Will give different *Function Signature* than regular reference…

What if…

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

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

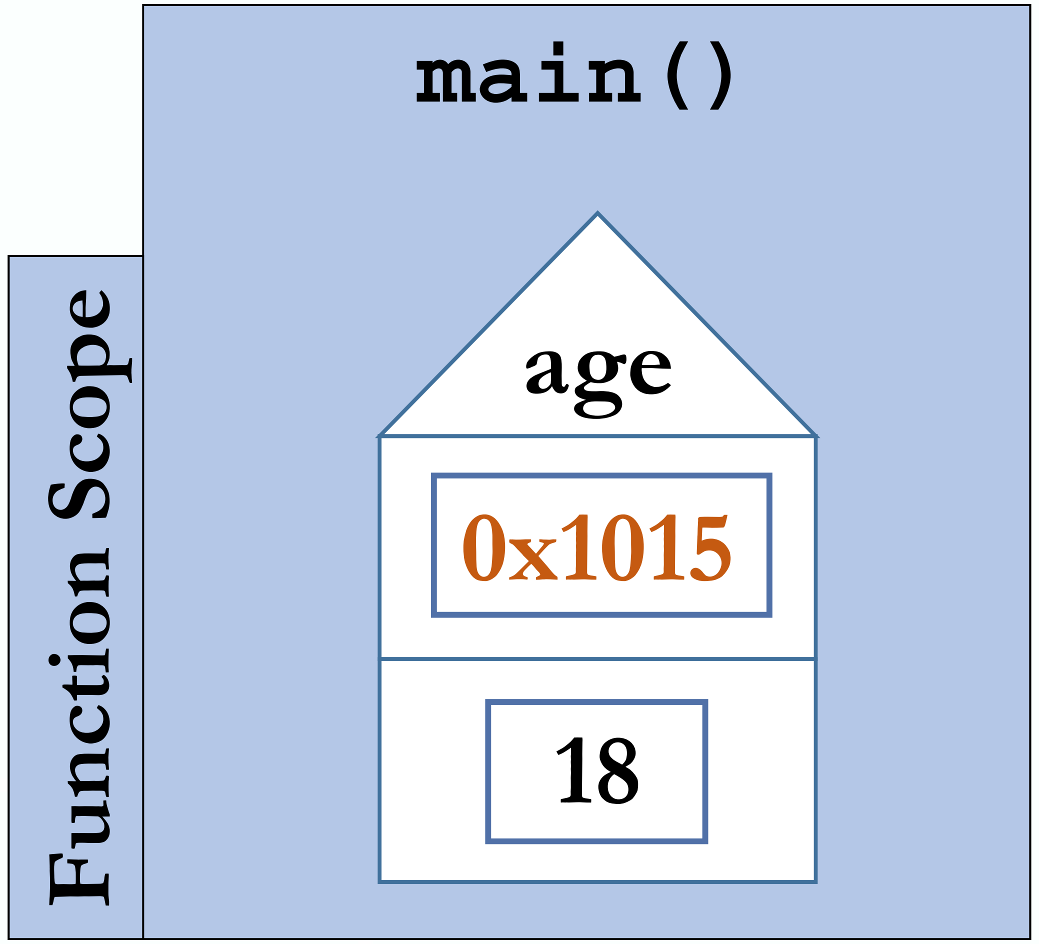
Pointers

Addresses

Remember Call-by-Value example:

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

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



Addresses commonly HEX numbers

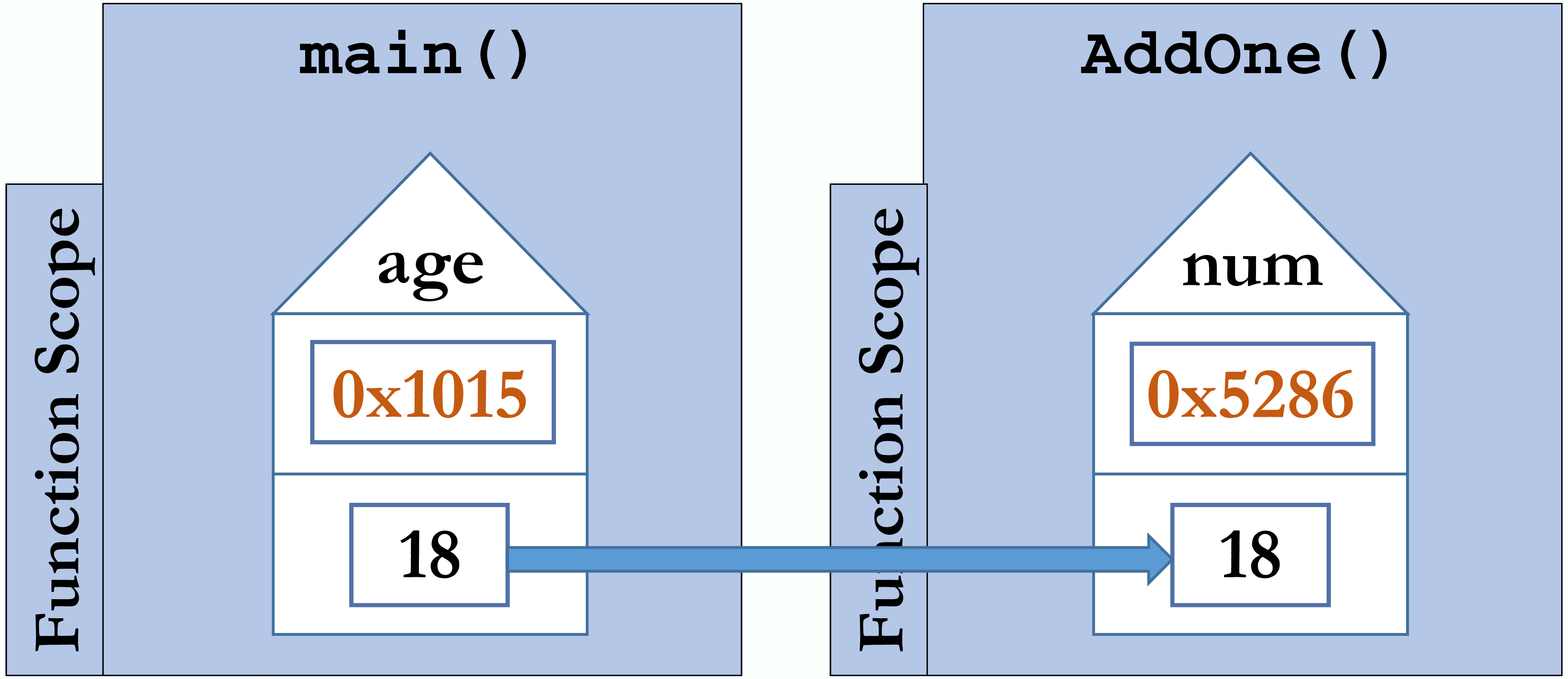
Pointers

Addresses

Remember Call-by-Value example:

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

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



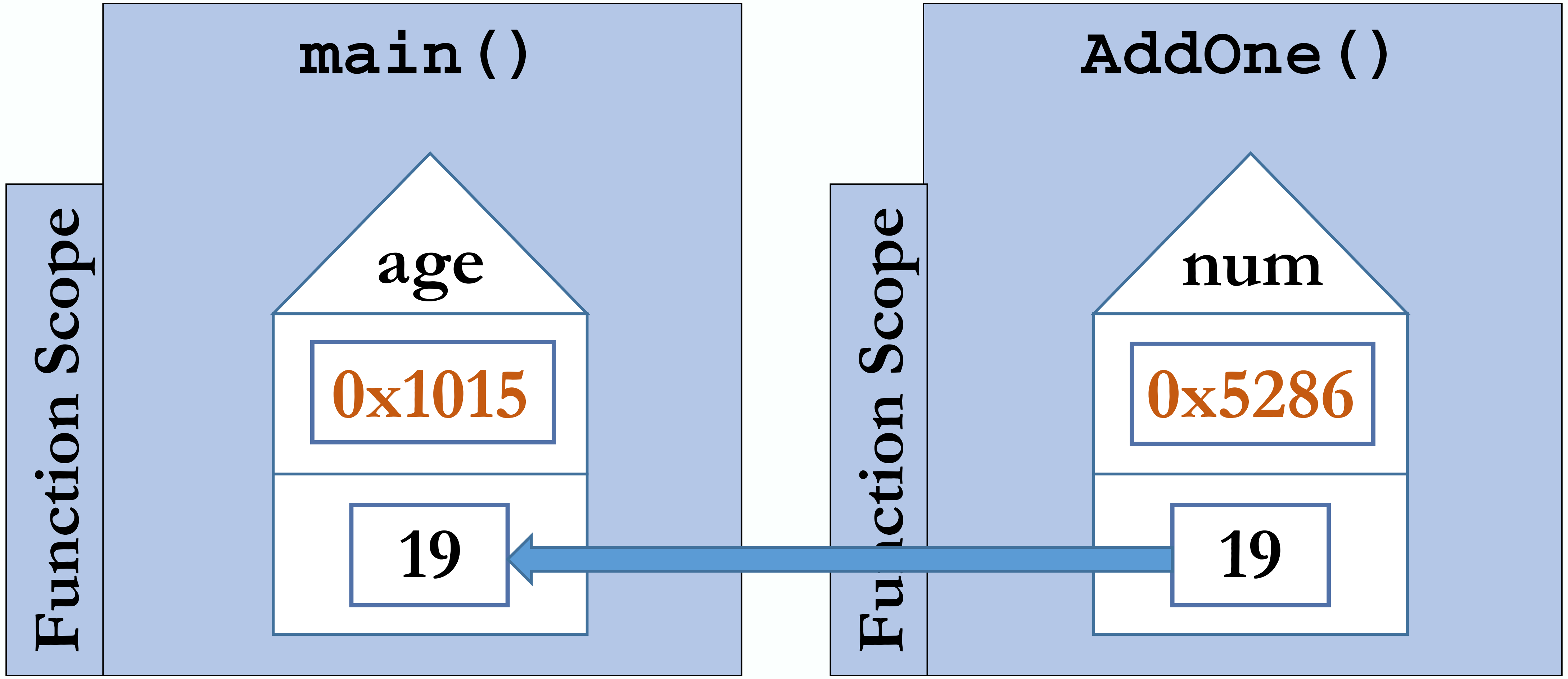
Addresses commonly HEX numbers

Addresses

Update via **return** value and assignment:

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

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



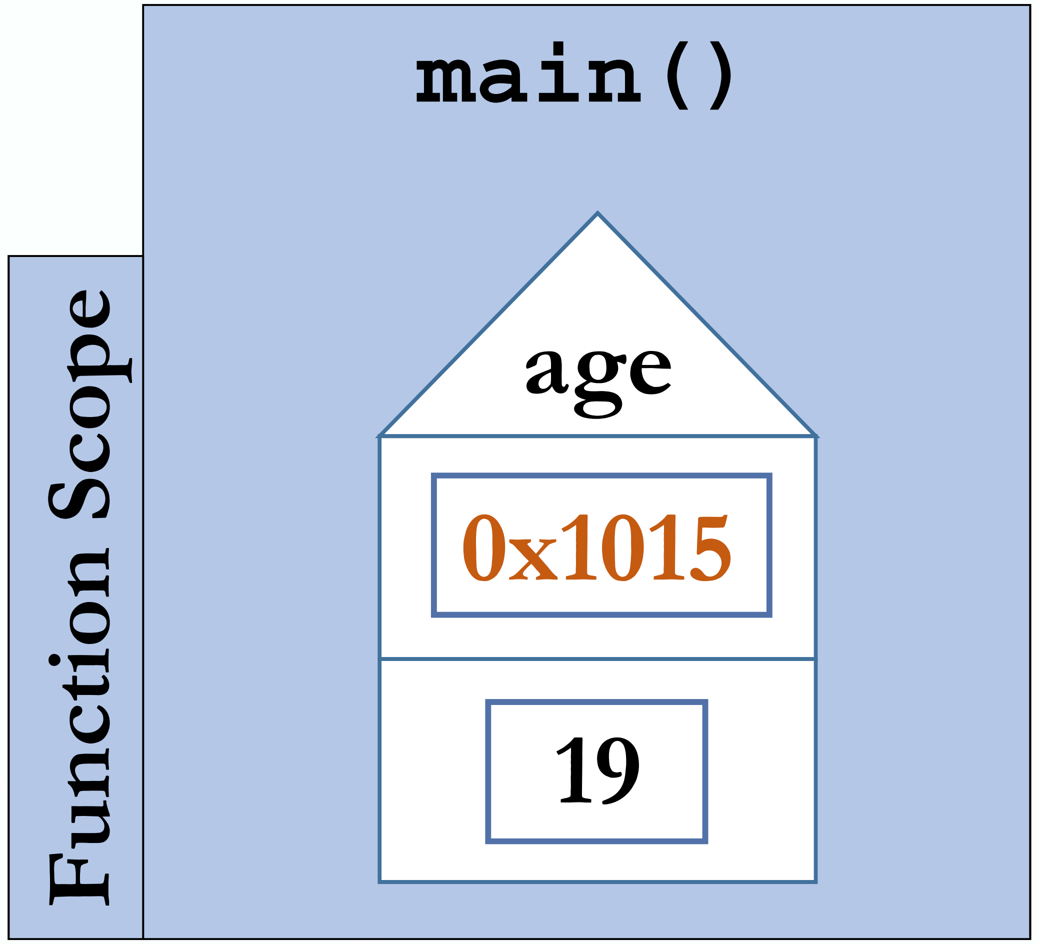
age, **num** in separate Scopes.

Addresses

After Function Call:

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

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

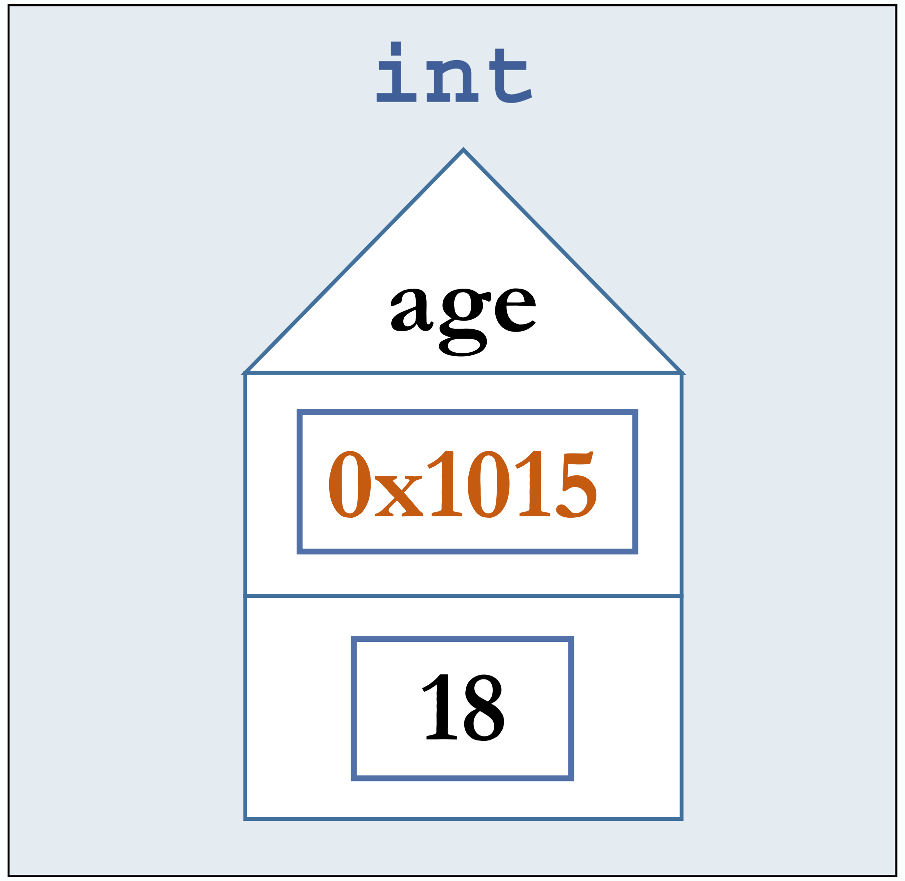


num not reachable outside its Scope Block (would not be available even if it were a static variable, it is not a life-time issue)

"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
```

Pointers

Pointer

A *Variable* whose Value represents an *Address-Of* something somewhere in memory.

```
int x = 37;
int *ptr;
cout << "x   is " << x << endl;
cout << "ptr is " << ptr << endl;
```

This will print out something like:

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

Addresses
commonly
HEX numbers

Pointers

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).

Pointers

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

Multiple pointers inline declaration / creation:

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

Pointers

Pointer Declaration

Valid pointers declaration / creation statements:

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

Avoid the last one.

Note:

- Multiple pointers inline declaration / creation:

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

int* ptr1, ptr2, ptr3;
```

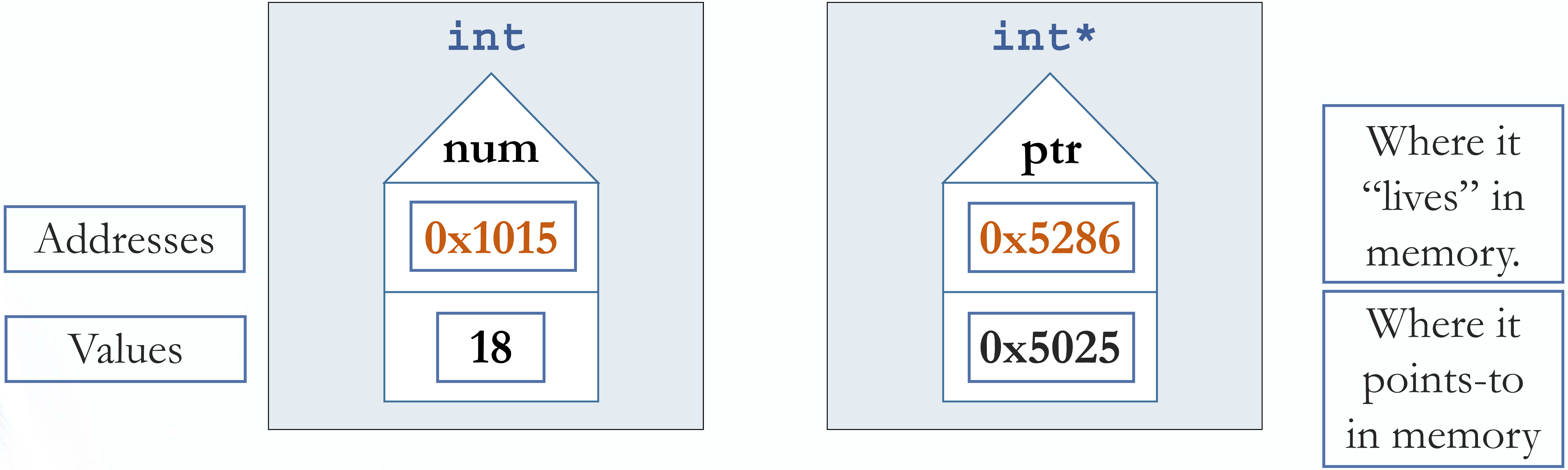
Looks right, but no.

Pointers

Pointer Value

As earlier stated, pointers are "Just Variables".

- ➢ Value: an Address in memory (instead of storing an `int`/`float`/`char`/etc.)



Pointers

Pointer Assignment

Value (pointed-to Address) assignment:

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

```
int  x = 5;
int *xPtr;
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.

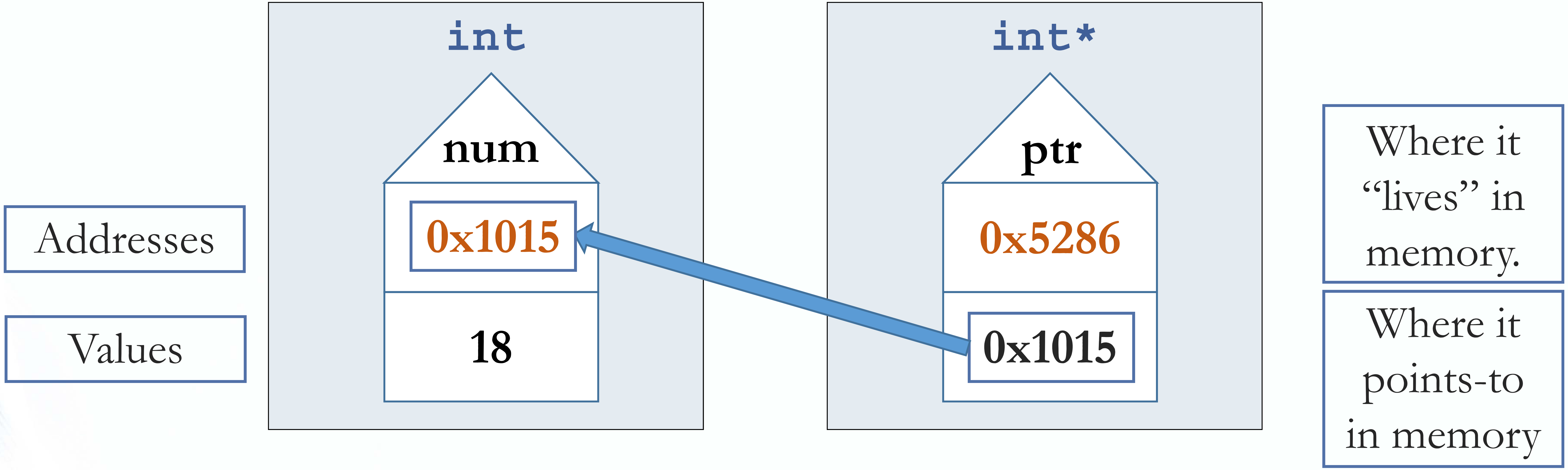
Compiler error: `"error: cannot convert 'int*' to 'char*' in initialization."`

Pointers

Pointer Assignment

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

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

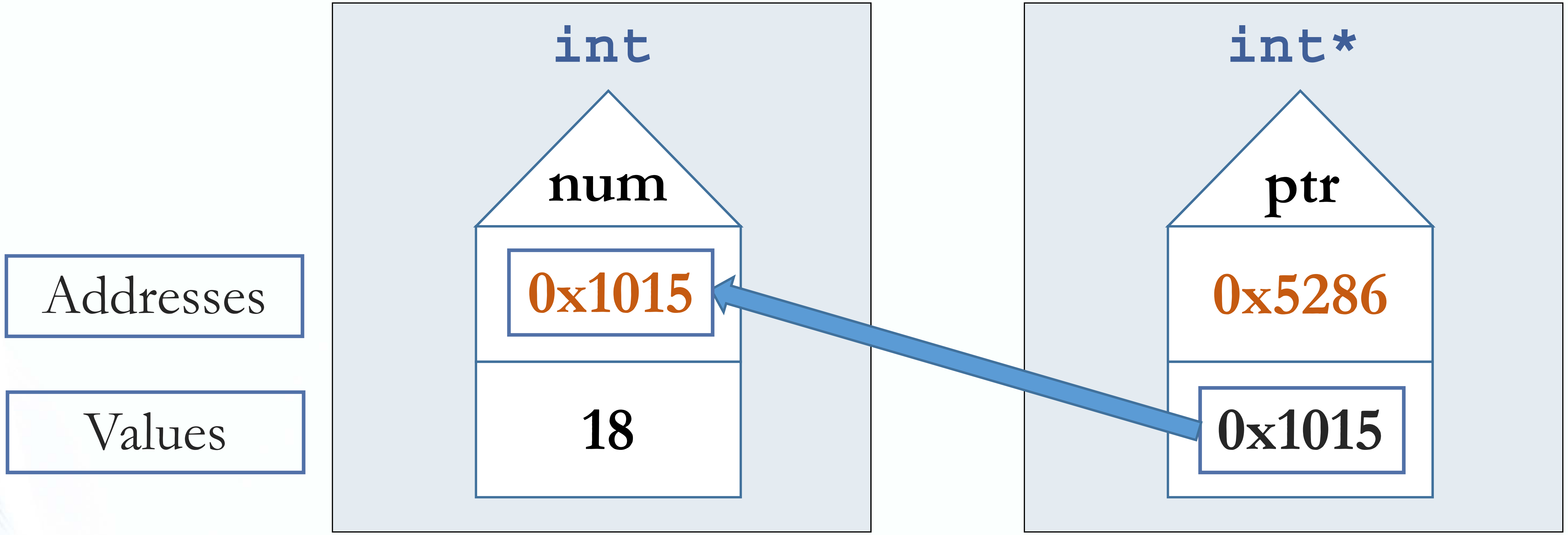


Pointers

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

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

```
… = *ptr
*ptr = …;
```

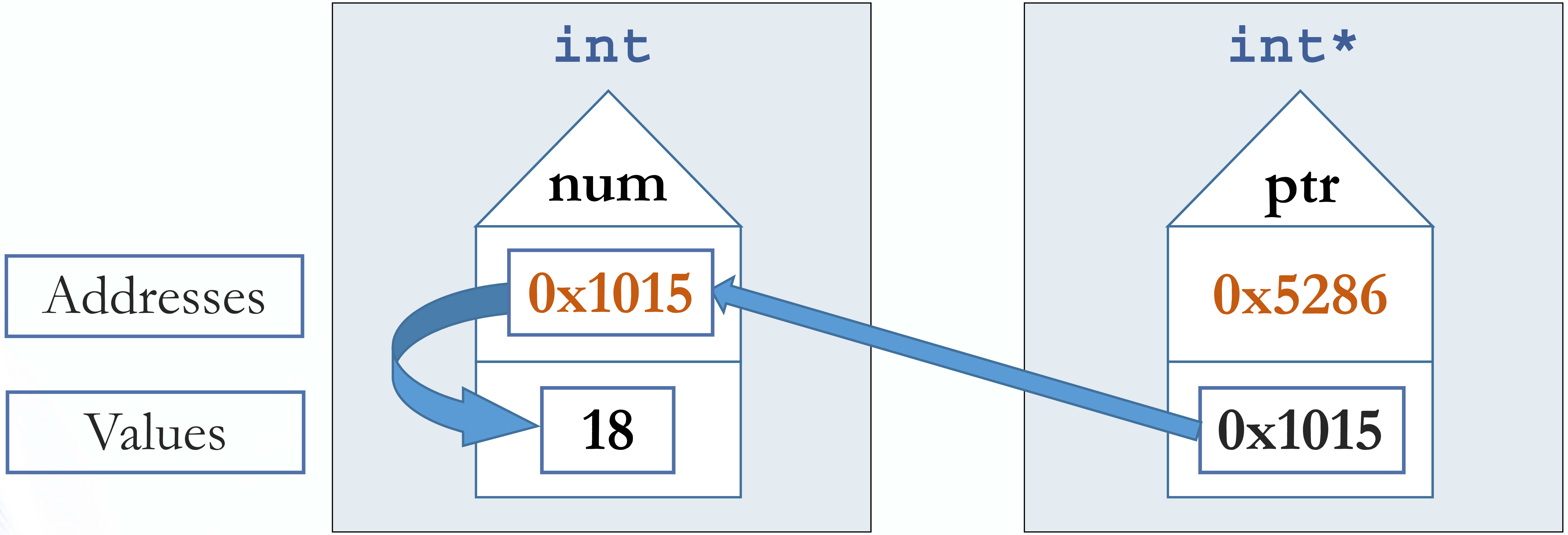


Pointers

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

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

```
… = *ptr
*ptr = …;
```



Pointers

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

At this point what follows depends on purpose of Dereferencing.

A Dereference can be in three "places":

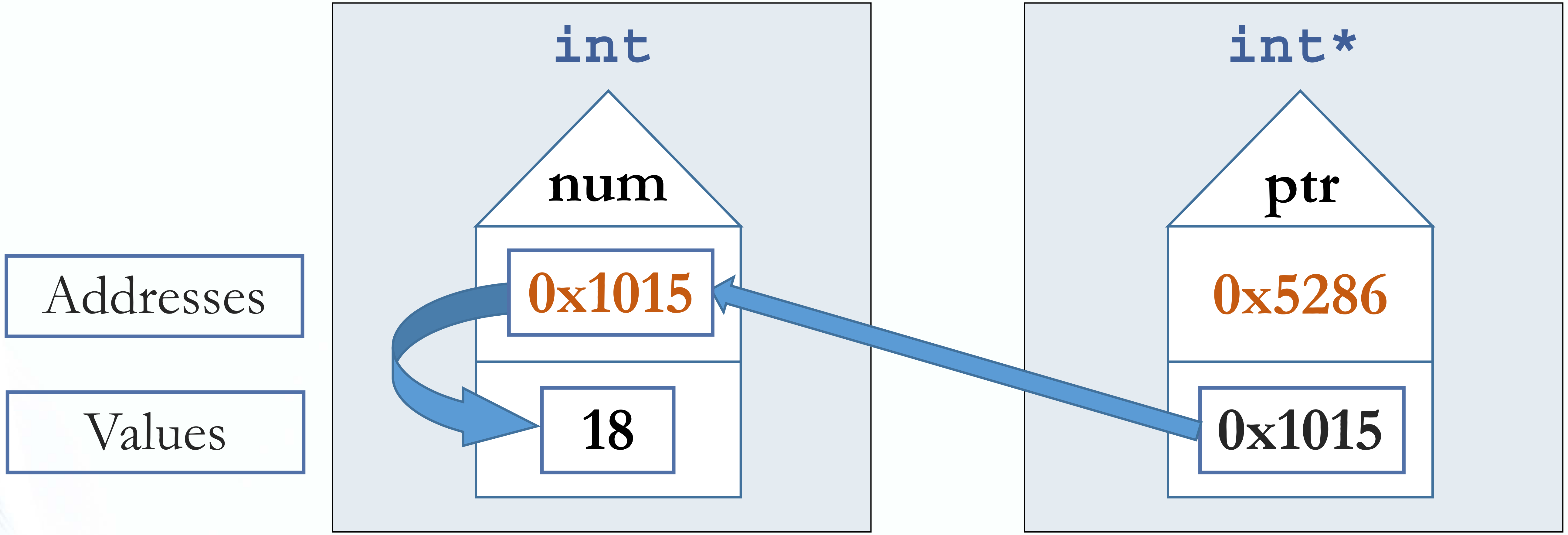
- On the *left hand* side of the assignment operator.
- On the *right hand* side of the assignment operator.
- In an expression with *no assignment* operator (e.g. a `cout` statement).

Pointers

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

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

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



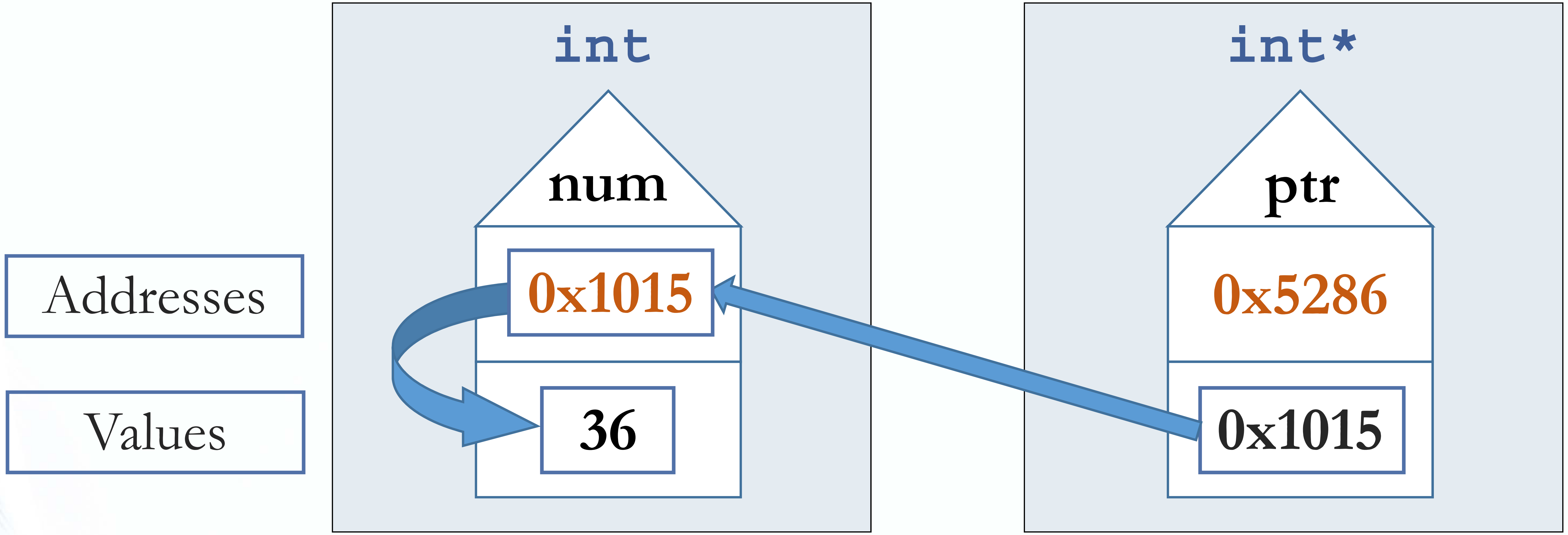
Access variable.
Get its value.

Pointers

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

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

```
*ptr = 36;
```



Access variable.
Change its value.

Pointers

Pointer Parameters in Functions

Common Paradgim:

- 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 Call-by-Value, Return, Assign method.
  `return` will only give back One value.

Pointer Parameters in Functions

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
}
```

Pointers

Pointer Parameters in Functions

Function Call:

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

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);
```

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*)

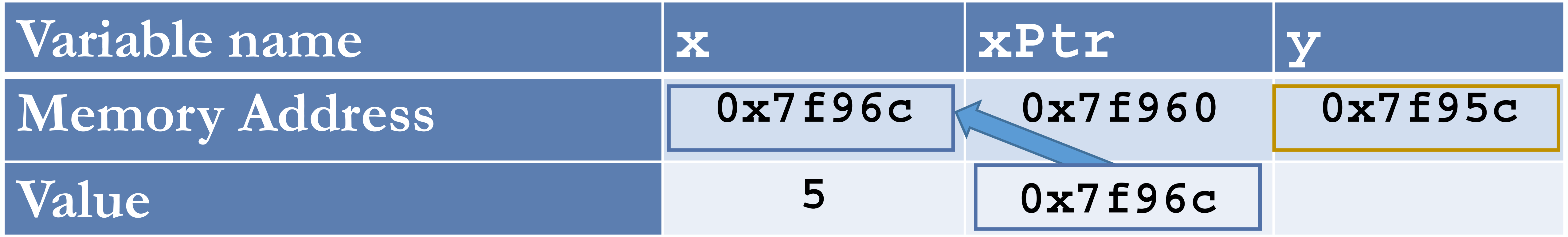
```
int  x = 5;
int *xPtr = &x;  /* xPtr points to x */
int  y = *xPtr;  /* y's value is now ... */
```

Variable name	x	xPtr	y
Memory Address	0x7f96c	0x7f960	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 ... */
```

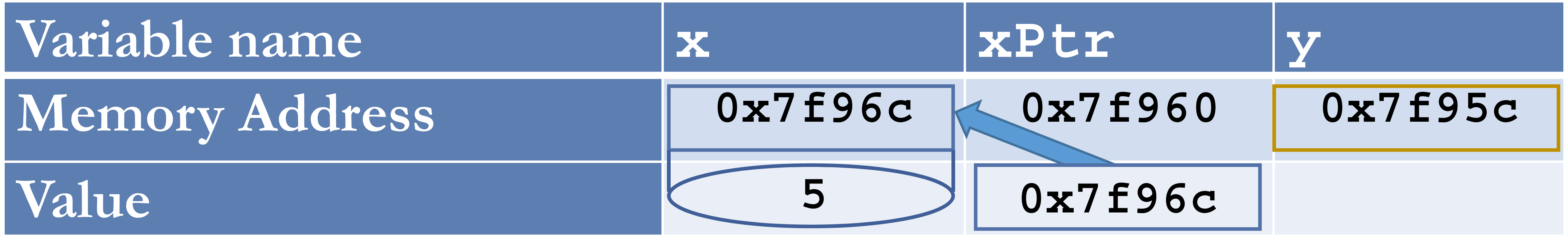


Variable name	x	xPtr	y
Memory Address	0x7f96c	0x7f960	0x7f95c
Value	5	0x7f96c	

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 ... */
```

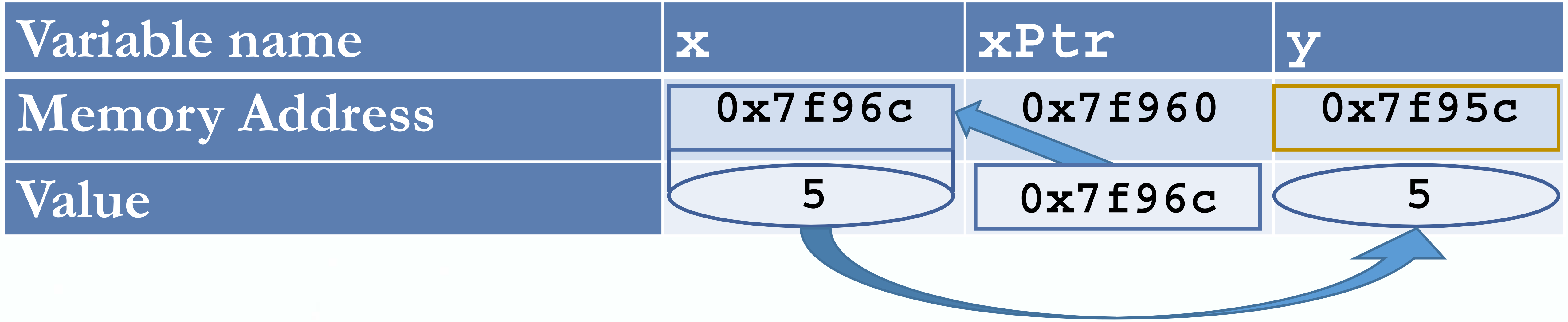


Variable name	x	xPtr	y
Memory Address	0x7f96c	0x7f960	0x7f95c
Value	5	0x7f96c	

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

References

Reference Declaration

Reference declaration the ampersand (&) symbol.

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

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

- They *look* like "normal" variables

Rules:

- References *must* be initialized at declaration.
  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.
- References can be treated as another "name" for a variable (no dereferencing).

References

Reference Declaration

Reference declaration the ampersand (&) symbol.

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

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

- 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.

References

Reference Caveats

Reference declaration the ampersand (&) symbol.

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

➢ Using them looks identical to using a value (Is easier always a good thing?) May think you're passing by value…

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

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

Valid Calls

Output:

```
changeByRef 2
changeByRef 2
main 1
```

References

Reference Caveats

Reference declaration the ampersand (&) symbol.

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

- ➢ Using them looks identical to using a value (Is easier always a good thing?) May think you're passing by value…

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

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

Valid Calls

Output:

```
changeByRef 2
changeByRef 3
main 3
```

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, *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, *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 !

Overview: Parameters in Functions

iii) Pass-by-Reference

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

```
void changeByRef (int &x);

int x = 1;
int &xRef = x;
```

```
changeByRef(x);
changeByRef(xRef);
```

Valid Calls

Overview: Parameters in Functions

iii) Pass-by-Reference

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

```
void changeByRef (int &x);

int x = 1;
int &xRef = x;
```

```
changeByRef(x);
changeByRef(xRef);
```

Valid Calls

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

Functions and Arrays

Pass-by-Address

Arrays are *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] = {};
```

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

Valid Definitions

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

by-Address
by-Reference (1st element)

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!";
```

or

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

Valid Definitions

or

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

by-Address
by-Reference (1st element)

CS-202

Time for Questions !