

CS222: Problem Solving With Programming II

B.Sc. Program

Winter 2019

Chapter I: Modular programming

In this chapter ...

- Modules and modularity
- Modularity in C++: Functions
- Scope of identifiers
- Inter module communication
- Storage classes: [Students' Independent work](#)

Module

- A module is a sequence of instructions that accomplishes a simpler and manageable task.

Module

- Definition of a module vs definition of a program
- Module and task
- Simpler & manageable task
- Module and size

Modular Program

- A modular program is a collection of interrelated modules.

Modular Program: Benefits

- Readability
- Independent & parallel development
 - Both coding & testing
- Team work facility
 - Division of labour
 - Accountability
- Maintenance facility
- Reusability

Modular Program ...

- Modularity solves scalability problem
 - Developing big programs
- Challenge: breaking down a big problem into its “logical” and “natural” components.
 - → a design issue!

Modular Program ...

- Modularity is about communication
- Modularity is first and foremost a design issue

Modularity in programming Languages

- How is the concept of “module” implemented in different programming languages?
 - Subroutines
 - Subprograms
 - Functions (C++, Java, and others)
 - Procedures
 - Classes/Objects (C++, Java, and others)

Modularity in C++: Functions

- In C++:
 - Functions
 - Classes/Objects: a course on OOP using Java → next term.

Function

- Function is a C++ “instrument” that implements a module.
- Hence,
 - a C++ function is a sequence of instructions that accomplishes a (simpler and manageable) task.
 - A C++ modular program is a collection of inter-communicating functions.

Function

- **Function definition/declaration**
 - Introducing the function into the program
 - In a program, a function can only be defined once.
- **Function invocation/calling**
 - Invocation/calling means using the function by invoking its name at appropriate point(s) in the “program”.
 - Once defined, a function may be called as many number of times as may be needed.

Function Definition

- Syntax:

< void > | <type> <fct_Idf> ([<formal parameters>])

<statement block>

- Note: the reded part is called the function heading or the function interface.

- Description:

- <fct_Idf> is the name of the function through which the function is invoked/called. The function may be given the stated parameters (if any) during calling.

Function Definition: Example

```
int nbOcc(char * txt, char ch)
{ int count = 0;
  int len = strlen(txt);
  for (; len >= 0; len--)
    if (txt [len] == ch) count++;
  return count;
}
```

Function Definition: Same Interface but different code

```
int nbOcc(char * txt, char ch)
{
    int count = 0;
    while (txt != NULL) {
        txt = strchr (txt, ch);
        if (txt != NULL) {
            count++;
            txt++;
        }
    }
    return count;
}
```

Note: Although the code is different, the “interface” remains the same.

Function Invocation/Calling

- Syntax:
 <fct_Idf> ([<actual parameters>])
- Description:
 - Call the function whose name is <fct_Idf> by transmitting the given actual parameters (if any).
- Note: Function calling is an expression.

Function Invocation: Example

```
int main()
{
    cout << nbOcc("Hello HiLCoE", 'i'); // Calling nbOcc() from
    within main()
    return 0;
}
```

Formal parameters

- Formal parameters
 - These are dummy arguments considered to be place holders. They will be substituted by the corresponding actual parameters when the function is called.
- Syntax
 - Comma separated `<var_idf>` declaration.

Actual parameters

- Actual parameters
 - These are the real arguments on which the function is executed at an instance of invocation/calling.
- Syntax
 - Comma separated constant values.
 - Each actual parameter is an expression.

Note

- The formal and actual parameters must usually match in the **order**, **type** and **number** of arguments.
- Exc: What is the implication if they don't match in the number (& type) of arguments?

Example

```
int fact (int n)
{ int i, p = 1;
  for(i=1; i<=n; i++)
    p = p*i;
  return p;
}

int main()
{ int m, f;
  cout << "Enter a natural number > ";
  cin >> m;
  // f = fact(m);
  cout << "The factorial is > " << fact(m) << endl;
  return 0;
}
```

Scope of Identifiers

- The scope of an identifier (name) is the part of the program in which the identifier is known.
- [For which the compiler doesn't say "undefined symbol" or any other similar error reports.]

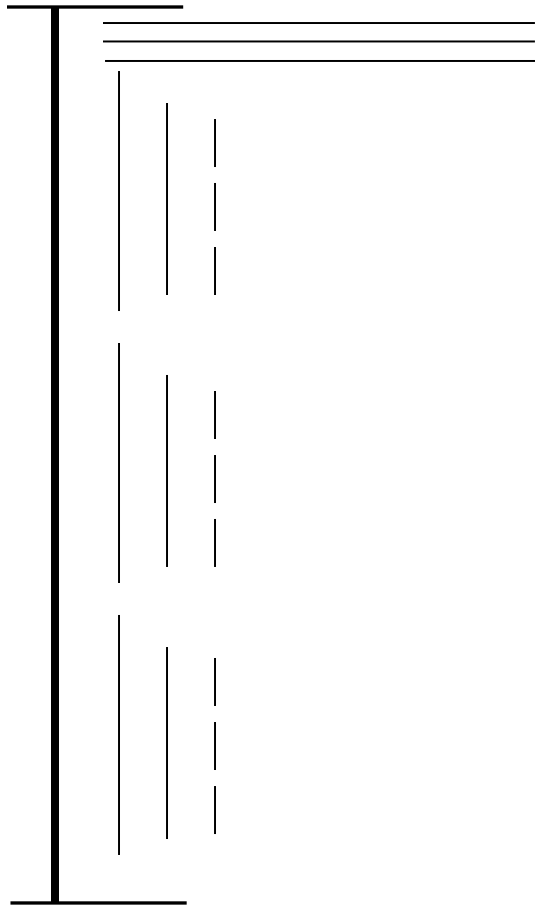
Exercise: Any thing wrong with the following arrangement?

```
int main()
{ int n, f;
  cout << "Enter a natural number > ";
  cin >> n;
  f = fact(n);
  cout << "The factorial is > " << f << endl;
return 0;
}
int fact (int n)
{ int i, p = 1;
  for(i=1; i<=n; i++)
    p = p*i;
  return p;
}
```

The Four Scopes

- Program Scope (File)
- Function Scope
- Block Scope
- Function Prototype Scope

Organization of the scopes



Organization of the scopes

- Program Scope
 - Function scope
 - Function prototype scope
- Function Scope
 - Block scope
- Block Scope
 - Block scopes

The Four Scopes

- Note: The following is not allowed in C++
 - A block on top of a program
 - A function on top of a function
 - The list of functions form a linear structure in C++.
 - A function prototype on top of a function
 - A function prototype on top of a block

Scope Rules

- Scope Rule 1
 - An identifier is known only within the scope in which it is declared and starting from its point of declaration down to the end of the scope.

Example: Any thing wrong?

```
int main()
{ int n, f;
  cout << "Enter a natural number > ";
  cin >> n;
  f = fact(n);
  cout << "The factorial is > " << f << endl;
  return 0;
}
int fact (int n)
{ int i, p = 1;
  for(i=1; i<=n; i++)
    p = p*i;
  return p;
}
```

The identifier fact() is used
out of its scope.

Example

```
void fact (int n, int & p) // for the alias operator &, see "Inter-module  
    comm"  
{ int i;  
  p = 1;  
  for(i=1; i<=n; i++)  
    p = p*i;  
}  
int main()  
{ int n, p;  
  cout << "Enter a natural number > ";  
  cin >> n;  
  fact(n, p);  
  cout << "The factorial is > " << p << endl;  
  return 0;  
}
```

The identifiers n and p are
declared in both functions.

Scope Rule 1: Advantages

- “Freedom” in using same identifiers (if need be) in different modules.
 - → facilitates team work

Example: Any thing wrong?

```
int n;  
int fact ()  
{ int i, p = 1;  
  for(i=1; i<=n; i++)  
    p = p*i;  
  return p;  
}  
int main()  
{ int f;  
  cout << "Enter a natural number > ";  
  cin >> n;  
  cout << "The factorial of " << n << " is " << f << endl;  
  f = fact();  
  return 0;  
}
```

The main() can access the
variable n even if n is not local to
it!

Example: When a function is void!

```
int n, p;  
void fact ()  
{ int i;  
  p = 1;  
  for(i=1; i<=n; i++)  
    p = p*i;  
}  
int main()  
{  
  cout << "Enter a natural number > ";  
  cin >> n;  
  fact();  
  cout << "The factorial is > " << p << endl;  
  return 0;  
}
```

The identifiers n and p are
common to both functions.

Exercise: What about now?

```
int n;  
int fact ()  
{ int i, p = 1;  
  for(i=1; i<=n; i++)  
    p = p*i;  
  return p;  
}  
int main()  
{ int n, f;  
  cout << "Enter a natural number > ";  
  cin >> n;  
  f = fact();  
  cout << "The factorial is > " << f << endl;  
  return 0;  
}
```

The variable n is re-declared
within the main() function

Scope Rules

- Scope Rule 2
 - If an identifier is re-declared within a nested scope, it is the inner declaration that takes precedence over the outer one.

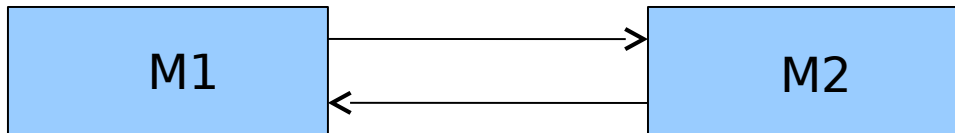
Exercise:

```
int n;  
int fact ()  
{ int i, p = 1;  
  for(i=1; i<=n; i++)  
    p = p*i;  
  return p;  
}  
int main()  
{ int n, f;  
  cout << "Enter a natural number > ";  
  cin >> n;  
  f = fact();  
  cout << "The factorial is > " << f << endl;  
  return 0;  
}
```

The variable n is re-declared within the main() function. This program does not accomplish the stated task.

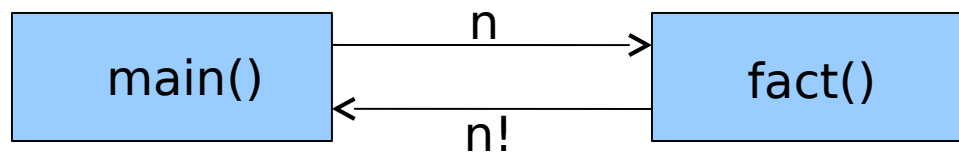
Inter-Module Communication

- It is about interactions between modules
 - ➔ Internal communication



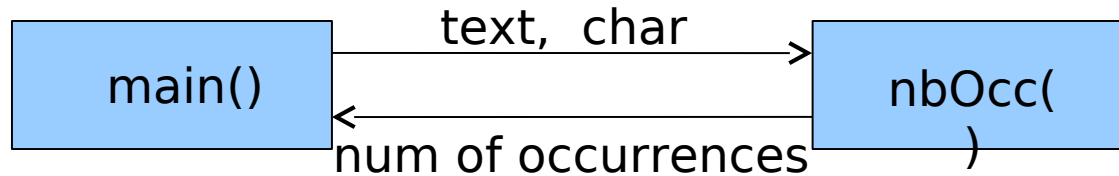
Inter-Module Communication

- Example: `main()` and `fact()`



Inter-Module Communication

- Example: main() and nbOcc()



Inter-Module Communication

- `void errorMessage1()`
- `{`
- `cout << "File not found";`
- `}`

Inter-Module Communication

- Example: `main()` and `errorMessage1()`



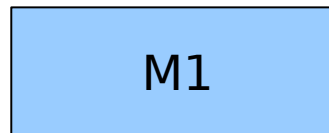
- No data transfer between the modules.

Inter-Module Communication

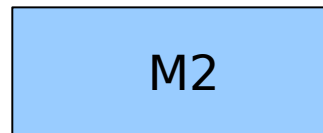
- Two types of communication:
 - Communication without data transfer
 - Calling (only)
 - No data transfer in both directions (eg. `errorMessage1()`)
 - Communication with data transfer
 - Calling
 - Data transfer

Protocol Channels

- Rules determining the format and transmission of data
- Three data transfer protocol channels: P1, P2, P3



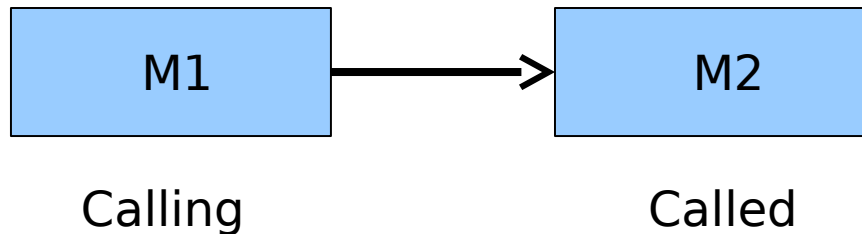
Calling



Called

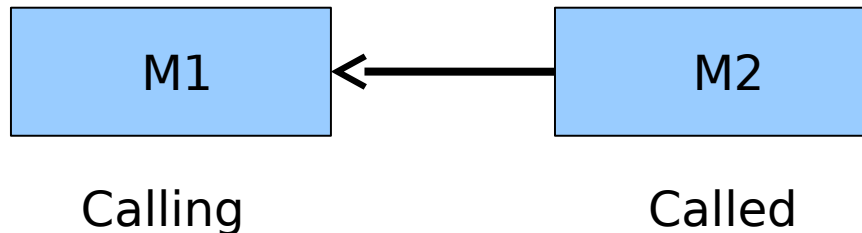
P1: Protocol Channel 1

- Data is moving from calling to called. This protocol channel is employed to bring data from the calling to the (called) function. This channel can not be used to return data back to the calling function.



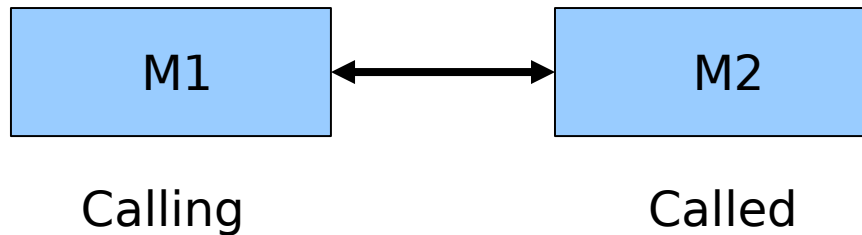
P2: Protocol Channel 2

- Data is moving from called to calling. This protocol channel is employed to send data from the (called) function to the calling function. This channel can not be used to bring data from the calling function.



P3: Protocol Channel 3

- Data is moving in both directions. This protocol channel allows data to be transferred in both directions.



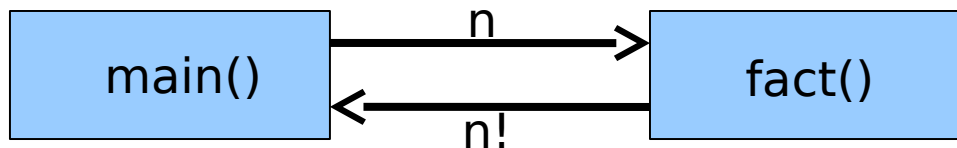
Protocol Channels: Notes

- In many programming languages, a protocol channel is implemented in such a way that it can only be used to transfer one type of data (int, int[], float, float[], char, char *, etc).
- Two communicating modules may open one or several protocol channels.

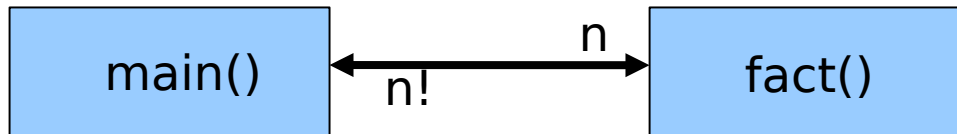
Protocol Channels: Example 1

- Two possibilities:

- Using P1 & P2



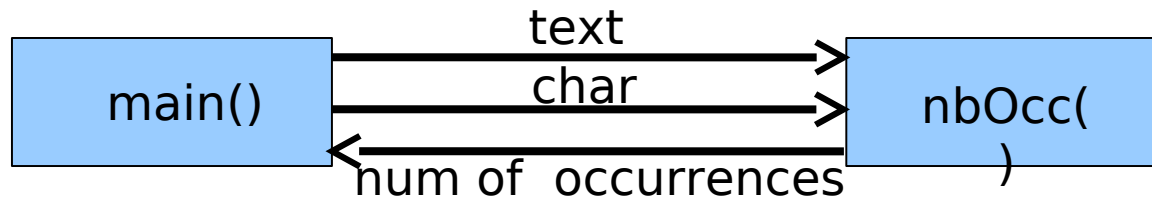
- Using P3



- Note: the second solution is not that interesting from a modularity aspect (see future slides).

Protocol Channels: Example 2

- One solution:
 - Using P1, P1 & P2



- Since all communicating data are of different type, they each require independent protocol channel.

Implementation of Protocol Channels(Tools)

- the means whereby some act is accomplished/or achieved
- How are the different (theoretical) protocol channels implemented in C++?
- Three data transfer (implementation) tools: T1, T2, T3

M1

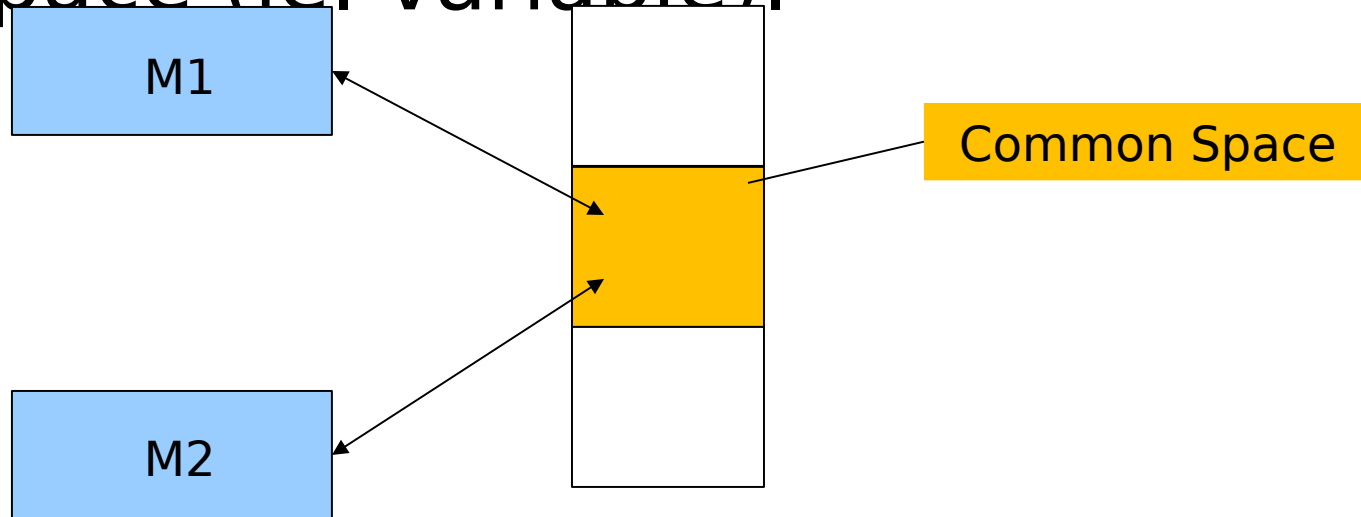
M2

Calling

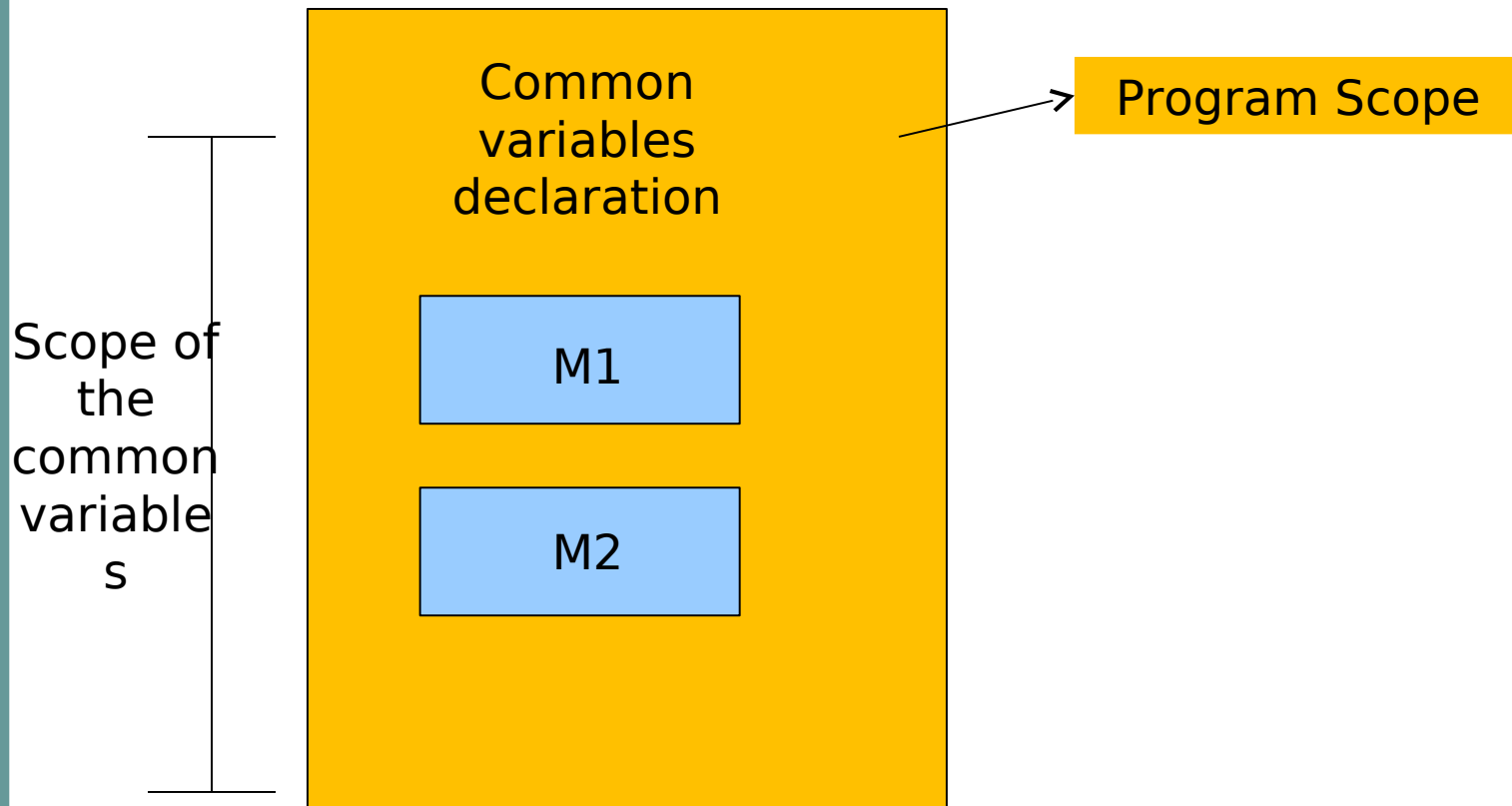
Called

T1: Data transfer using common variables

- Two communicating modules can exchange data if they have a commonly accessible memory space (ie. variable).



T1: Data transfer using common variables



T1 Example 1 - On P1 & P2

```
int n, p;    // common variables
void fact () // no formal parameters and function not typed!
{ int i,
  p = 1; // p is not declared locally
  for(i=1; i<=n; i++) // n is not declared locally
    p = p*i;
} // no return statement

void main()
{ cout << "Enter a natural number > ";
  cin >> n; // not declared locally
  fact (); // no actual parameters
  cout << "The factorial is > " << p << endl; // not declared locally
} Note: the variable i is not used for data transfer and hence is local to fact().
```

T1 Exercise 1 - On P3

Implement Example 1 on P3.

T1: Answer to the exercise - On P3

```
int n;    // common variable
void fact () // no formal parameters and function not typed!
{ int i, p;
  p = 1; // p is now declared locally
  for(i=1; i<=n; i++) // n is not declared locally
    p = p*i;
n = p; // Since p is not known within main(), the result must be copied
      to n.
} // no return statement
void main()
{ cout << "Enter a natural number > ";
  cin >> n; // not declared locally
  fact (); // no actual parameters
  cout << "The factorial is > " << n << endl;
    // the variable n contains both the number and its factorial!
} Note: This programming practice is to be avoided! Why?
```

T1 Exercise 2 - On P3

- Implement Example 1 on P3 but using the variable p as a common variable.

T1: Example 2

```
char * txt, ch;      // common variables
int count; // common variable
void nbOcc() // no formal; function is not typed!
{
    count = 0;
    while (txt != NULL) {
        txt = strchr (txt, ch);
        if (txt != NULL) { count++; txt++; }
    };
} // no return statement
int main()
{
    strcpy(txt, "Hello World");
    ch = 'l';
    nbOcc(); // no actual
    cout << "Number of occurrences = " << count << endl;
    return 0; // this return statement is only due to main()
}
```

T1: Protocols supported

- Although T1 could be used to implement P1, P2 and P3 protocols, it actually is a P3 tool.
- Why?

T1: Important Notes

- In C++, common variables can only be provided at the program scope.
- Such variables are commonly termed “global” variables.
- It is not a good programming practice to declare a variable “globally” if it is not used for data transfer between two communicating modules.

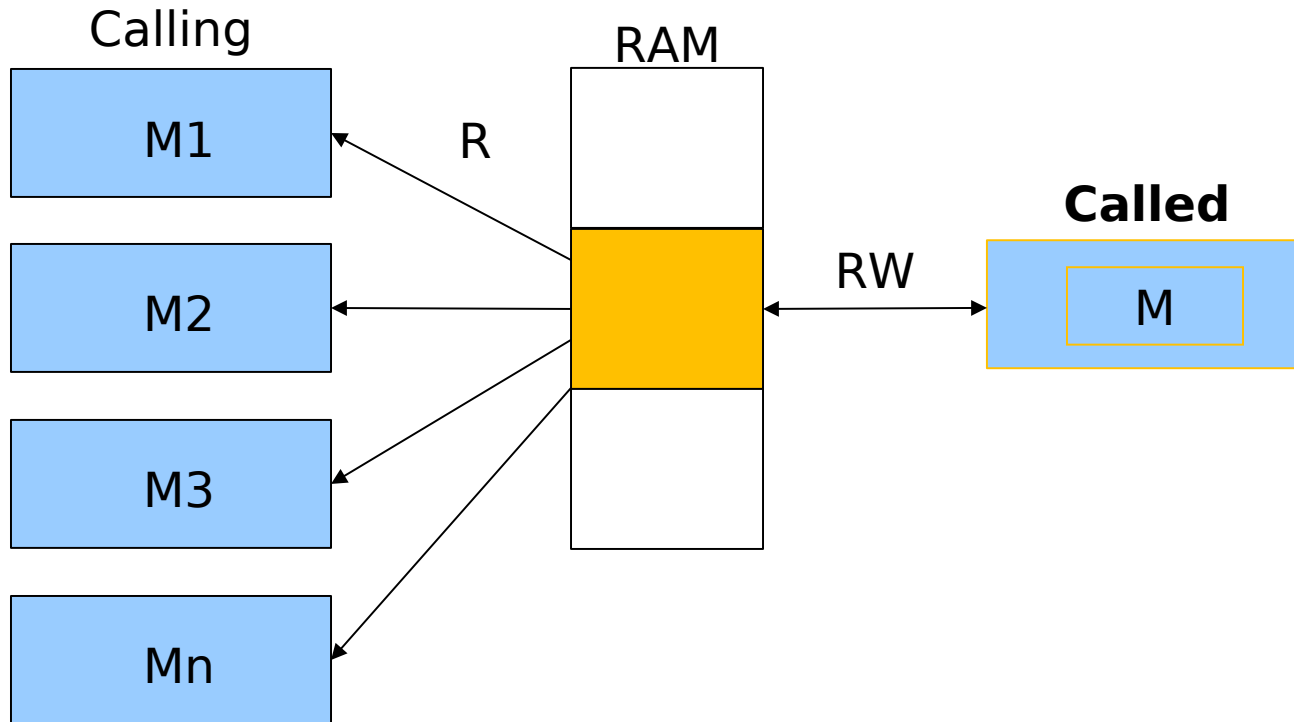
T2: Data transfer using the “return” statement

- May also be called “using the function identifier (name)”
- The function identifier is used for invoking the function.
- In this case, the function is given an additional role - that of carrying a data to transfer it back to the calling function.
- In other words, the function identifier acts like a variable.

T2: Data transfer using the “return” statement

- Implying that a memory space is reserved for the function identifier.
- Implying that the function must be typed (ie. not void).
- Such a space, however, is special in the sense that the function itself has a **Read/Write** access to it whereas all other functions could only have a **Read** access.
- A data is written to this memory space only using the return statement (from within the called function).

T2: Data transfer using the “return” statement



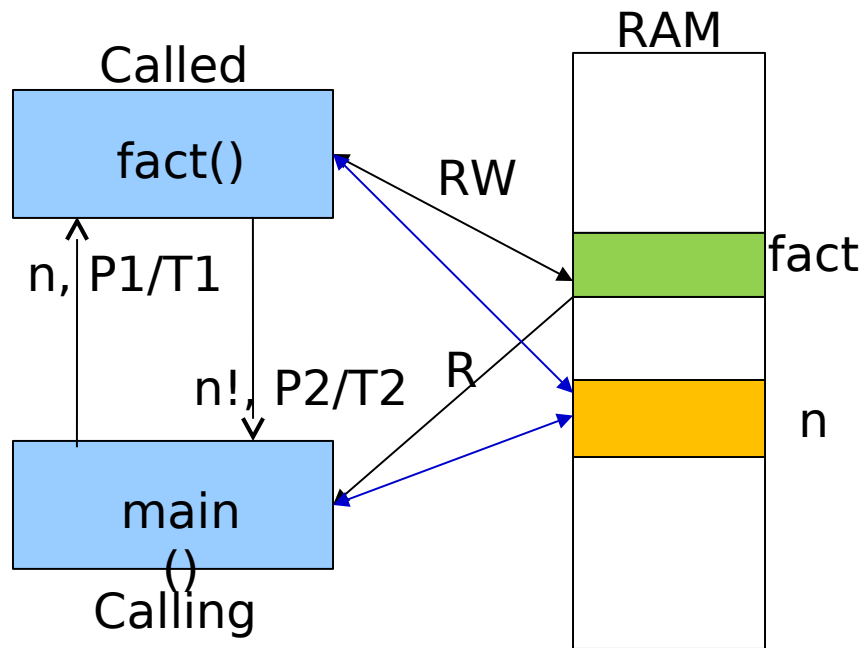
T2:Protocol(s) Supported

- Since only the called function has the privilege to write (and read) into the memory space reserved for the function identifier and only read access for the calling functions, data can only be transferred from the called to the calling.
- Implying that only P2 is supported by T2.

T2: Example 1

```
int n;  
int fact () // function is typed!  
{ int i, p;  
  p = 1;  
  for(i=1; i<=n; i++)  
    p = p*i;  
  return p; // 'return' copies the content of p into the space reserved for  
            'fact'.  
}  
void main()  
{ int f;  
  cout << "Enter a natural number > ";  
  cin >> n;  
  f = fact (); // Had fact() been void, the assignment would have been wrong!  
  cout << "The factorial is > " << f << endl; // or cout << "The ..." << fact()  
        << endl;;  
}
```

T2: Example 1



T2: Example 2

```
char * txt, ch;    // common variables
```

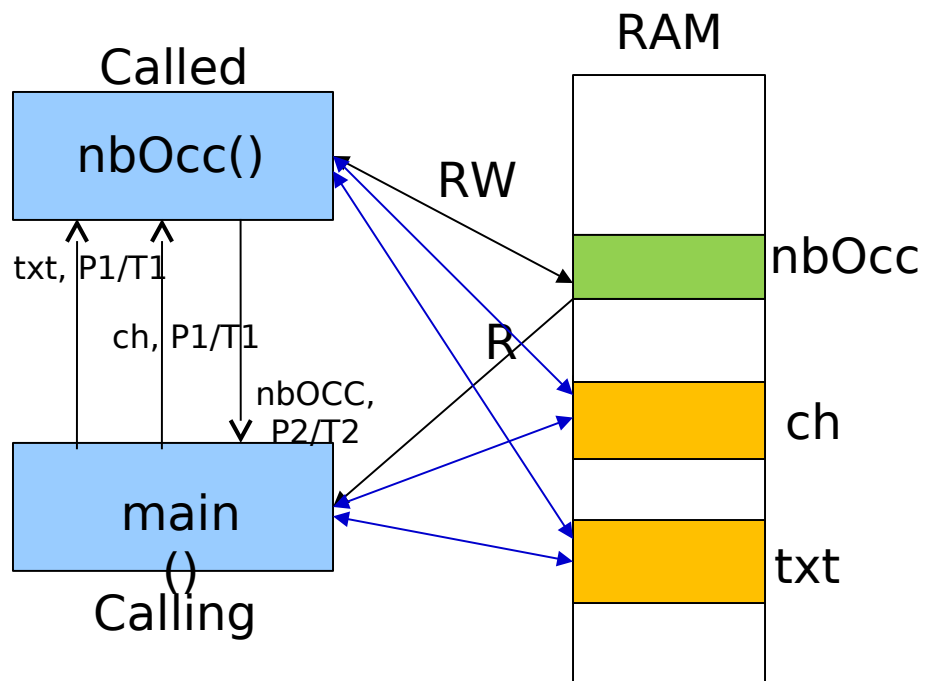
```
int nbOcc() // function is typed!
```

```
{ count = 0;
  while (txt != NULL) {
    txt = strchr (txt, ch);
    if (txt != NULL) { count++; txt++; }
  } return count;
}
```

```
int main()
```

```
{      cout << "Enter a line of text > "; cin.getline(txt, 100);
  cout << "Enter a character > "; cin >> ch;
  cout << "Number of occurrences = " << nbOcc() << endl;
  return 0; // this return statement is only due to main()
}
```

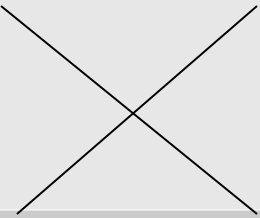
T2: Example 2



T2: Notes

- A function could only return a “single” data value thru the return statement.
- ➔ “Single” value means:
 - Simple types such as int, float, char, pointer, struct, etc.
 - Array and string are not included.
- However, there is a means to return array and string values in a return statement: how?

T3: Data transfer using parameter passing

Protocol Supported	Simple Type + Struct	Array + String
P1	default	Const
P2		
P3	Pointer & Alias	default

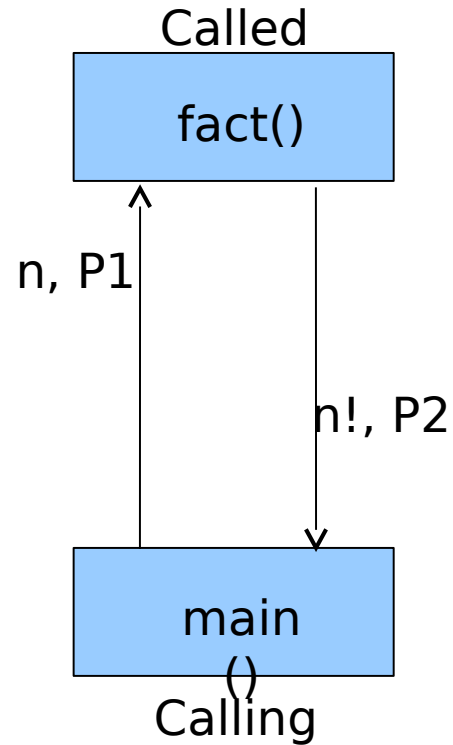
T3: Data transfer using parameter passing

- The term “default” is used in the sense that “pointer”, “alias” or “const” are not added in the formal parameters.
- P1 is provided by default if the parameter is of a simple type (“integers”, “reals”, char, “pointer”, “boolean”, enum) and struct type.
- P3 is provided by default if the parameter is an array or a string.
- If P3 is requested on a simple type and on a struct parameter, “pointer” or “alias” must be used.
- If P1 is requested on an array and string type, a **const** qualification must be used.
- P2 is not supported as such. However, one may use P3 in order to get P2.

T3: Data transfer using parameter passing

Simple type parameters.

T3: Example 1



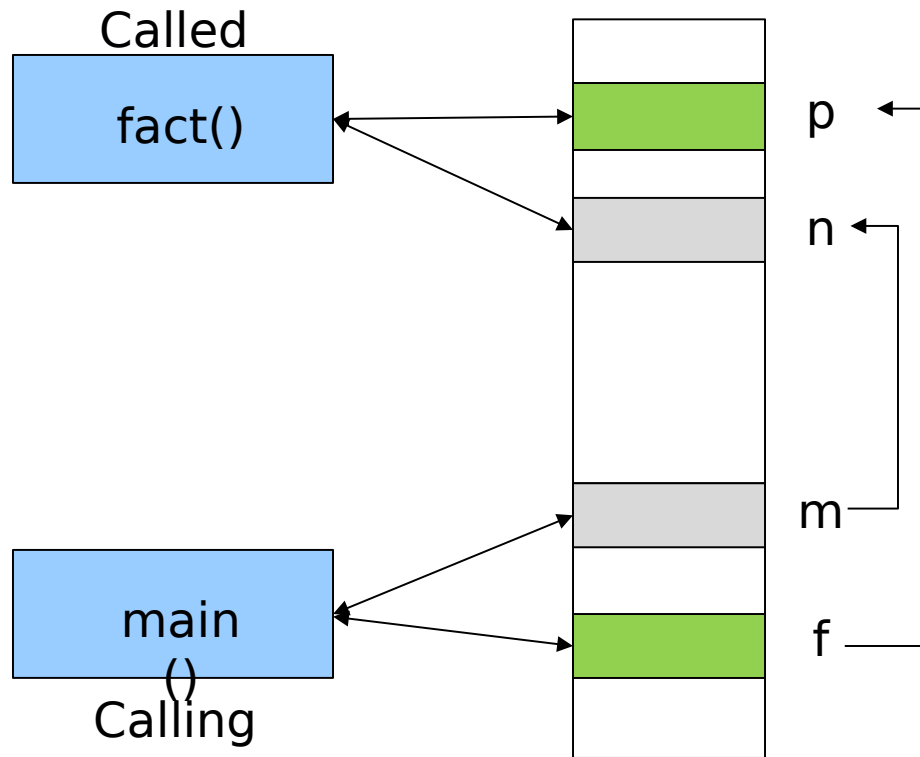
What type are the parameters?

T3: Is this program correct?

```
// no common variable
void fact (int n, int p) // formal parameters
{ int i;;
  p = 1;
  for(i=1; i<=n; i++)
    p = p*i;
} // no return statement

void main()
{ int m, f;
  cout << "Enter a natural number > ";
  cin >> m;
  fact (m, f); // actual parameters
  cout << "The factorial is > " << f << endl;
}
```

T3: Is this program correct? ...



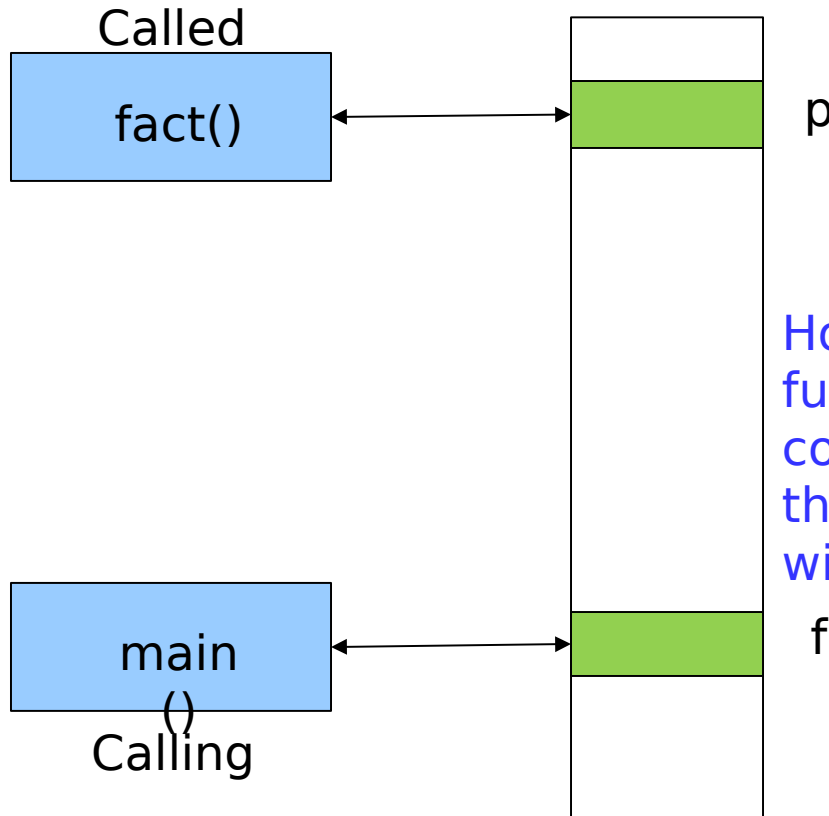
T3: Is this program correct? ...

- The communication need in the above program is P1 for the parameter m and P2 for the parameter p. Both parameters are simple types (both int).
- We remark that no **pointer** or **alias** is used in the formal parameters.
- Implying that the protocol provided is P1 in both parameters which is correct for parameter m but not for parameter p.
- If this program is executed, it will display a wrong result since the computed factorial could not be transferred back to the calling function (ie. main()).

T3: Is this program correct? ... Solution

- The solution is to revert the P1 provided for the parameter p by default into P2.
- How?
 - Using pointer
 - Using alias

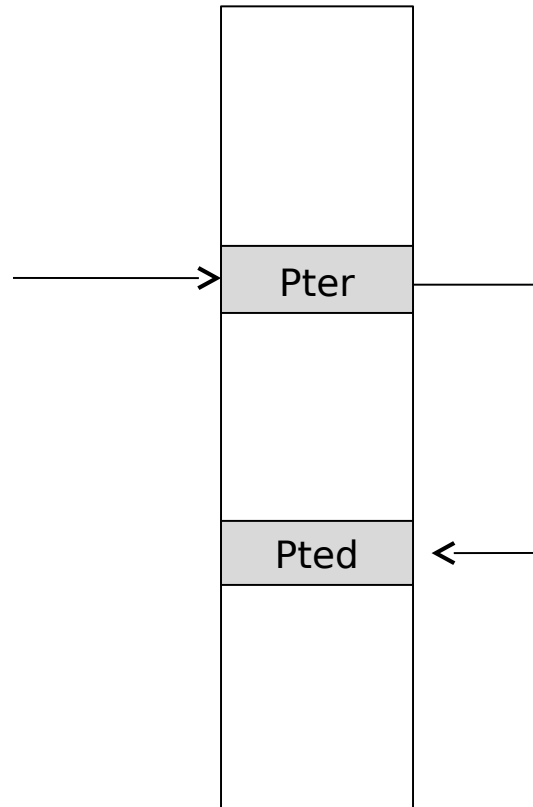
T3: Using pointer ...



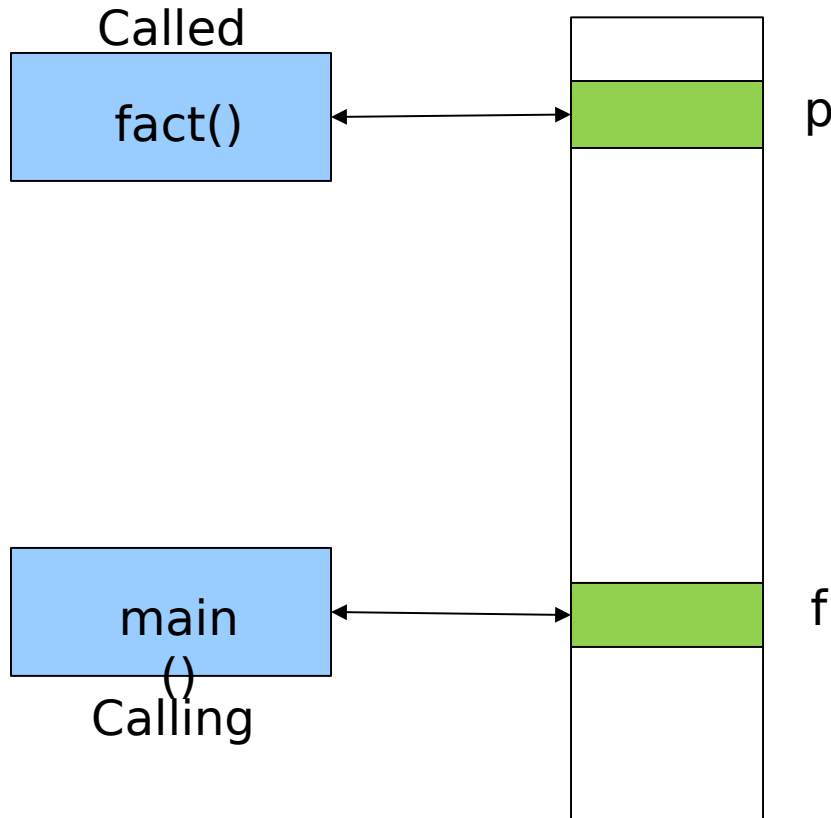
How is it possible for fact() function to write back the computed factorial back to f thru the parameter p, without directly accessing f?

T3: The pointer concept

Accessing a
certain space
indirectly!



T3: Using pointer ...



Which should be the
pointer and which
the pointed?

T3: Using pointer ...

- Two options:

1. f points to p

2. p points to f

Discuss the two options!

T3: Using pointer ...

One of the option can not even
be realized leaving the other
option the only possible
solution!

T3: Using pointer ...

```
// no common variable
void fact (int n, int * p) // formal parameters
{ int i, q;
  q= 1;
  for(i=1; i<=n; i++)
    q = q * i;
  *p = q;
} // no return statement
void main()
{ int m, f;
  cout << "Enter a natural number > ";
  cin >> m;
  fact (m, &f); // actual parameters
  cout << "The factorial is > " << f << endl;
}
```

Using Alias

T3: The alias concept

- An alias is another name
- In C++, an alias is created for variable identifiers as follows:
 - Syntax
 - `<type> & <idf> = <var-idf>;`
 - Description
 - `<idf>` another name (ie. alias) for `<var-idf>`.

T3: The alias concept

- Example
 - `float x = 6.8;`
 - `float & y = x;`
 - `x++;`
 - `y++;`
 - `cout << x; // ➔ 8.8`
 - `cout << y; // ➔ 8.8`

T3: The alias concept

- Note:
 - We can only define an alias for an existing variable identifier.

T3: Using alias ...

- Two options:
 1. f is alias for p
 2. p is alias for f

Discuss the two options!

T3: Using alias ...

One of the option can not even be realized leaving the other option the only possible solution!

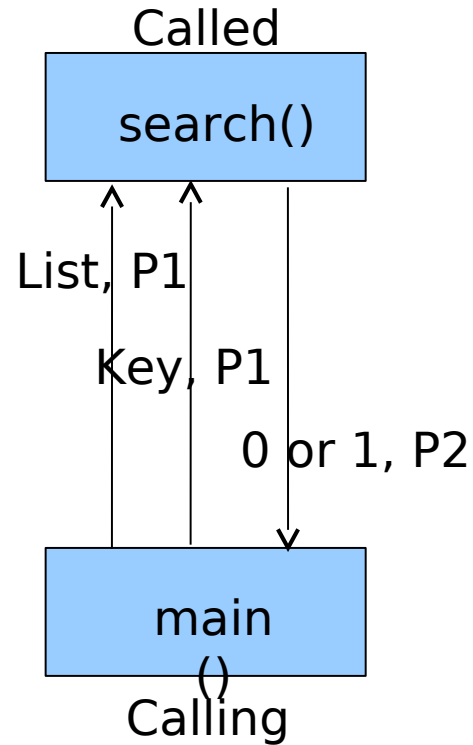
T3: Using alias ...

```
// no common variable
void fact (int n, int & p) // formal parameters
{ int i;;
  p= 1;
  for(i=1; i<=n; i++)
    p = p * i;
} // no return statement
void main()
{ int m, f;
  cout << "Enter a natural number > ";
  cin >> m;
  fact (m, f); // actual parameters
  cout << "The factorial is > " << f << endl;
}
```

T3: Data transfer using parameter passing

Array & string parameters.

T3: Example 2 – Searching in a list



T3: Example 2 ...

```
void main()
{ float b[100], sk;
  int m, i, found;
  cout << "Enter the size of your list > "; cin >> m;
  cout << "Enter the elements of the list one by one \n\
  n";
  for (i=0; i<=m-1; i++) cin >> b[i];
  cout << "Enter a search key > "; cin >> sk;
  search (b, m, sk, found);
  if (found) cout << sk << " is in the list " << endl;
  else cout << sk << " is not in the list" << endl
}
```

T3: Example 2 ...

```
void search (float a[100], int n, float sk, int &
    found)
{ int i;
  found = 0;
  i = 0;
  while(i<=n-1 && !found)
    if (a[i] == sk) found = 1;
    else i=i+1;
}
```

T3: Example 2 ... (T2 + T3)

```
int search (float * a, int n, float sk /*, int & found
*/)
{ int i, found;
  found = 0;
  i = 0;
  while(i<=n && !found)
    if (a[i] == sk) found = 1;
    else i=i+1;
  return found;
}
```

T3: Example 2 ... (T2 + T3)

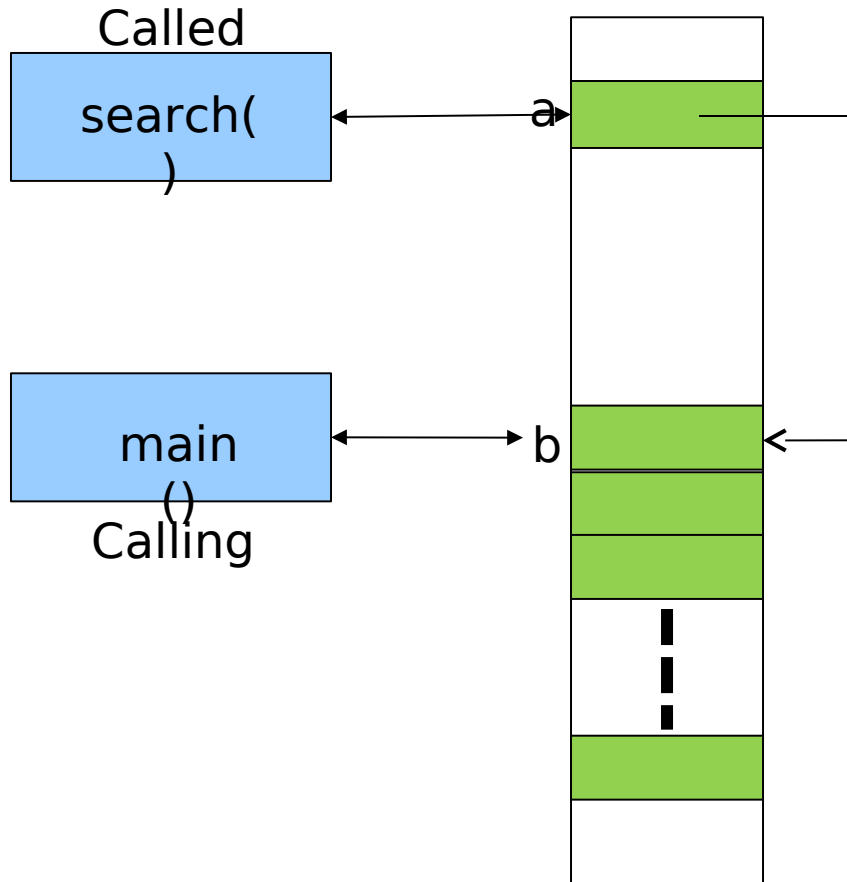
```
void main()
{ float b[100], sk;
  int m, i, found;
  cout << "Enter the size of your list > "; cin >> m;
  cout << "Enter the elements of the list one by one \n\
  n";
  for (i=0; i<=m-1; i++) cin >> b[i];
  cout << "Enter a search key > "; cin >> sk;
  found = search (b, m, sk /*, found*/);
  if (found) cout << sk << " is in the list " << endl;
  else cout << sk << " is not in the list" << endl
}
```

T3: Example 2 ... (T2 + T3)

```
int search (float * a, int n, float sk /*, int & found */)
{ int i, found;
  found = 0;
  i = 0;
  while(i<=n && !found)
    if (a[i] == sk) found = 1;
    else i=i+1;
  // what will be the effect of modifying a[ ] to the main?
  a[0] = -99.9;
  return found;
}
```

Could the function search() has the right to modify the array a[]? Should it be allowed to do so?

T3: Example 2 ... Mem Layout



T3: Example 2 - Using the const qualification

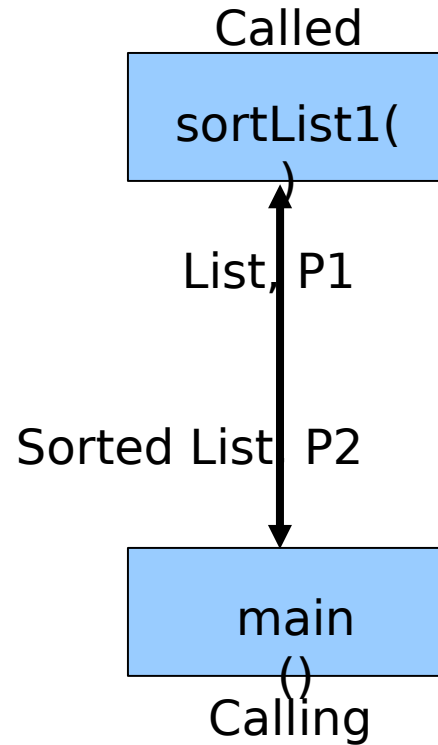
```
int search (const float * a, int n, float sk)
{ int i, found;
  found = 0;
  i = 0;
  while(i<=n && !found)
    if (a[i] == sk) found = 1;
    else i=i+1;
  // a[0] = -99.9; // what about now?
  return found;
}
```

What about now? Does the array a[] could still be modified?

T3: Example - a[] is the same as *

```
int search (const float a[ ], int n, float sk)
{ int i, found;
  found = 0;
  i = 0;
  while(i<=n && !found)
    if (a[i] == sk) found = 1;
    else i=i+1;
  return found;
}
```

T3: Example 3 - Sorting on P3



T3: Example 3 - Sorting on P3

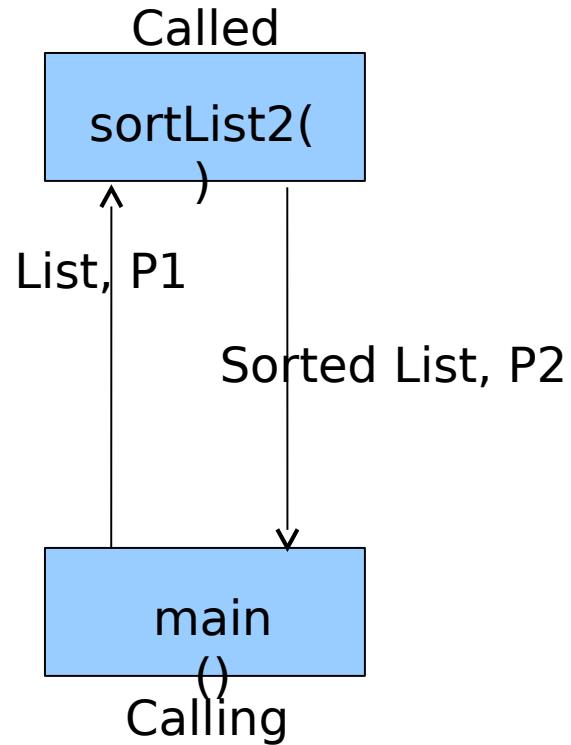
```
void sortList1 (float * a, int n)
{ float temp;
  for (int i = 1; i <= n-1; i++)
    for (int j = n-1; j >= i; j--)
      if (a[j] < a[j-1]) {
        temp = a[j];
        a[j] = a[j-1];
        a[j-1] = temp;
      }
}
```

T3: Example 3 - Sorting on P3

```
void main()
{ float b[100];
  int m, i;
  cout << "Enter the size of your list > "; cin >> m;

  cout << "Enter the elements of the list one by one
  \n\n";
  for (i=0; i<=m-1; i++) cin >> b[i];
  sortList1 (b, m);
  cout << "The list after sorting \n\n";
  for (i=0; i<=m-1; i++) cout << b[i] << " ";
}
```

T3: Example 3 - Sorting on P1 & P2



This is left as an exercise.

T3: Example 3 - Sorting on P1 & P2

```
void sortList2 (const float * a, float * c, int n)
{ float temp;
  for (i=0; i<=n-1; i++)
    c[i] = a[i]; // copy the array a[ ] into c[ ]
  for (int i = 1; i <= n-1; i++)
    for (int j = n-1; j >= i; j--)
      if (c[j] < c[j-1]) {
        temp = c[j];
        c[j] = c[j-1];
        c[j-1] = temp;
      }
}
```

T3: Example 3 - Sorting on P1 & P2

```
void main()
{ float b[100], p[100];
  int m, i;
  cout << "Enter the size of your list > "; cin >> m;

  cout << "Enter the elements of the list one by one
  \n\n";
  for (i=0; i<=m-1; i++) cin >> b[i];
  sortList2 (b, p, m);
  cout << "The list after sorting \n\n";
  for (i=0; i<=m-1; i++) cout << p[i] << " ";
}
```

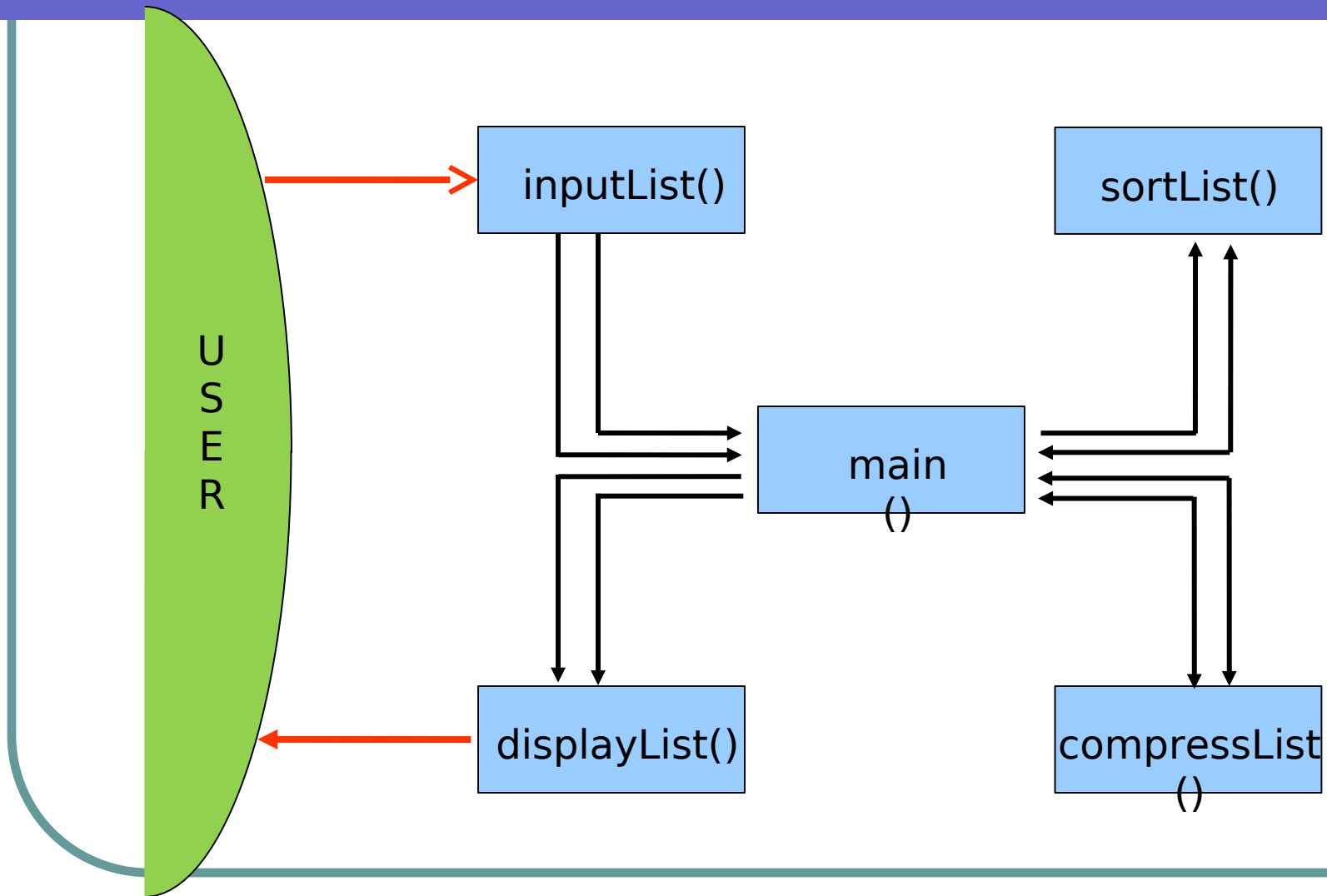
T3: Example 3 - Sorting on P1 & P2

```
void main() /* alternatively, main could copy the array
    locally and still use sortList1() */
{ float b[100], p[100];
  int m, i;
  cout << "Enter the size of your list > "; cin >> m;
  cout << "Enter the elements of the list one by one \n\n";
  for (i=0; i<=m-1; i++) cin >> b[i];
  for (i=0; i<=n-1; i++) p[i] = a[i]; // copy the array a[] into
    p[]
  sortList1 (p, m); // calling sortList1() - not sortList2()
  cout << "The list after sorting \n\n";
  for (i=0; i<=m-1; i++) cout << p[i] << " ";
}
```

T3: Example 4 – Compressing a sorted list

- Sorted List:
2,2,2,3,4,4,6,6,6,8,10,10,12,12,16,16,16,16
- Compressed List: 2,3,4,6,8,10,12,16

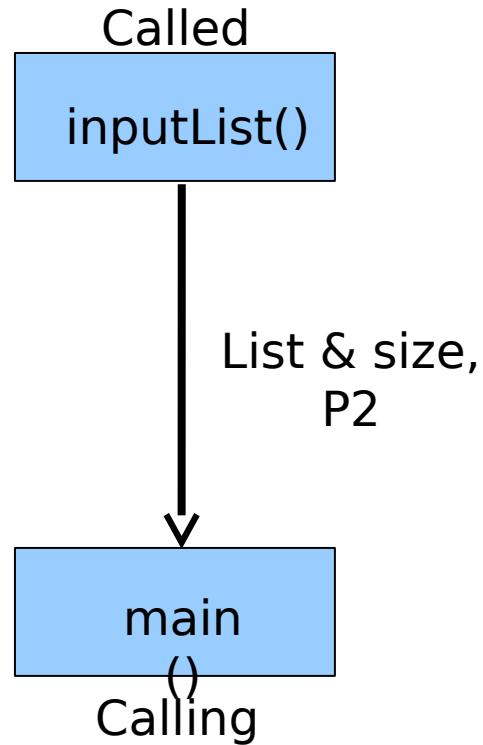
T3: Example 4 – Compressing a sorted array



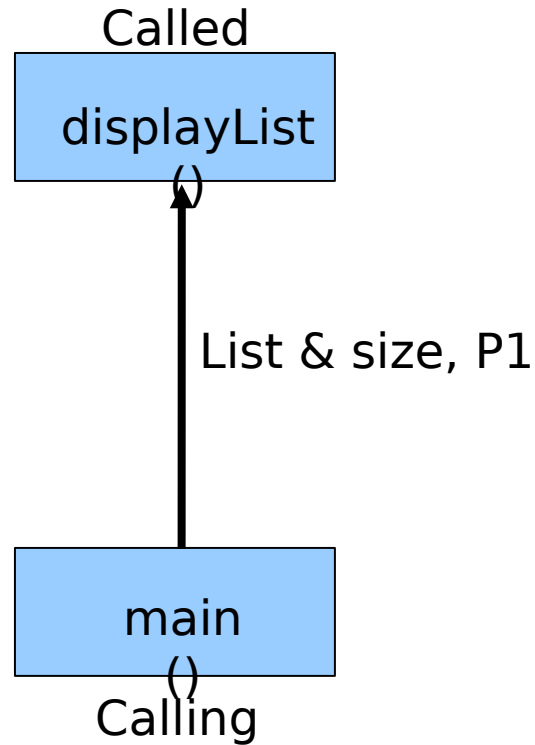
T3: Example 4 – the main() as we like it!

```
void main()
{ float b[100];
  int m;
  clrscr();
  inputList(b, m); // receives the list from the user
  sortList (b, m); // sort the list
  compressList (b, m); // compress the list
  displayList(b, m); // display the list
  getch();
}
```

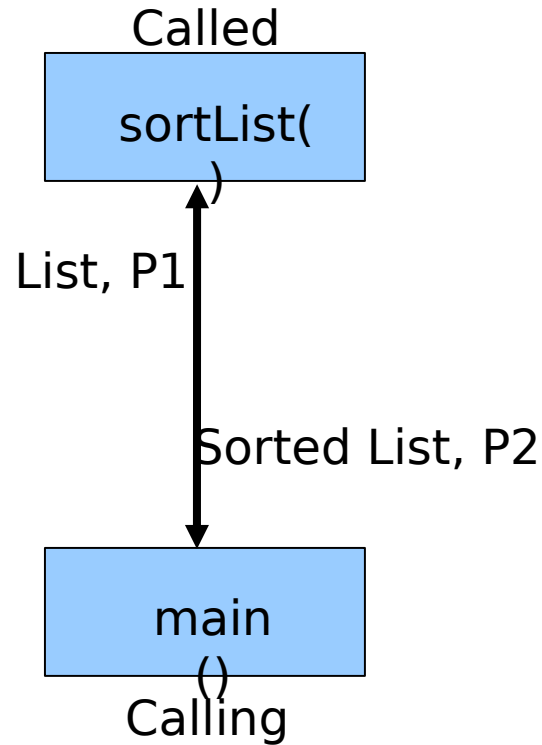
T3: Example 4 – InputList()



T3: Example 4 – displayList()

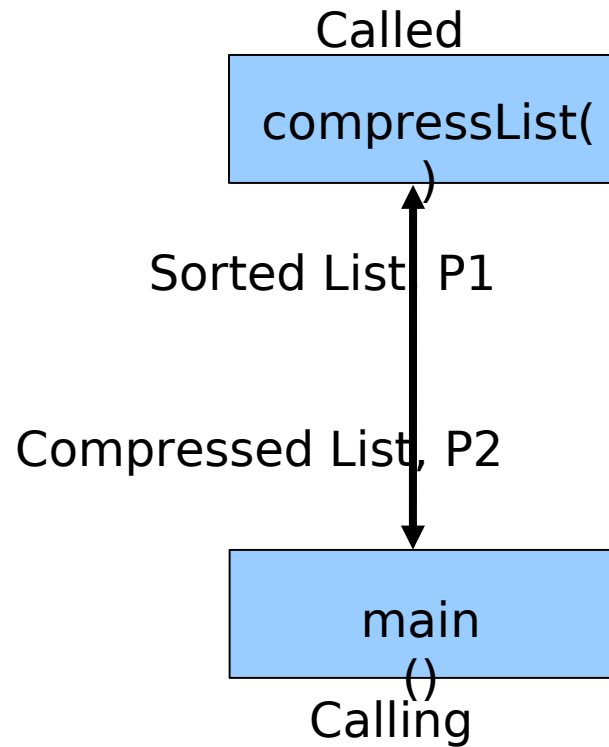


T3: Example 4 – sortList()



One P3 channel serving both P1 & P2.

T3: Example 4 – compressList()



One P3 channel serving both P1 & P2.

T3: Example 4 – inputList()

```
void inputList(float a[ ], int & m)
{
    cout << "Enter the size of the list > ";
    cin >> m;
    cout << "Enter the elements of the list one by
        one \n\n";
    for (int i=0; i<=m-1; i++) cin>> a[i];
}
```

T3: Example 4 – displayList()

```
void displayList(const float a[ ], int m)
{
    cout << "***** List size = " << m << " ***** "
        << endl;
    for (int i=0; i<=m-1; i++)
        cout << a[i] << " ";
    cout << endl << endl;
}
```

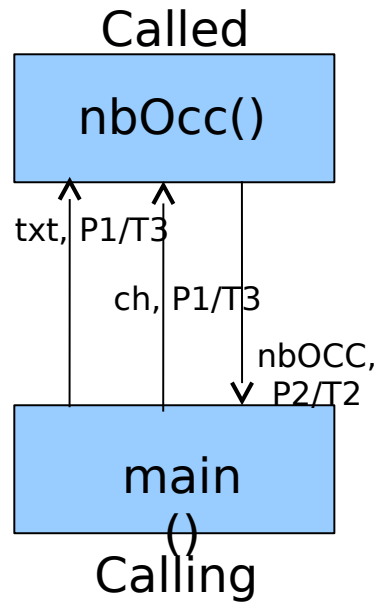
T3: Example 4 – sortList()

```
void sortList (float a[ ], int n)
{   float temp;
    for (int i = 1; i <= n-1; i++)
        for (int j = n-1; j >= i; j--)
            if (a[j] < a[j-1]) {
                temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
}
```

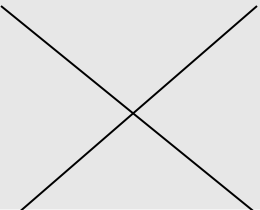
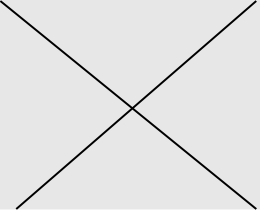
T3: Example 4 – compressList()

```
void compressList (float a[ ], int & n)
{ int i, j, dup;
  for (i = 0; i <= n-1; i++) {
    j=i;
    while (j+1<=n-1 && a[j+1] == a[i]) j++;
    dup= j - i; // number of xtra duplicates of current element
    if (dup > 0) {
      for (j=i+1; j<=n-1-dup; j++)
        a[j] = a[j+dup];
      n = n - dup;
    }
  }
}
```

T3: Exercise – Implement it using T2 & T3



T3 versus Parameter passing by value / by address

Protocol Supported	Simple Type + Struct	Array + String
P1	Default By Value	Const By Address
P2		
P3	Pointer & Alias By Address	Default By Address

T3: Data transfer using parameter passing

Simple type parameters.