Functions: C++ Programming Modules

Function Basics

To use C++ function:

- Prototype

```
void simple();
```

- Definition

```
void simple(){
    using namespace std;
    cout <<"Simple Function";
}
```

- Call the function

```
int main(){
    using namespace std;
    simple(); // call the function
}
```

General form of function definition

```
typeName functionName(parameterList){
    statements;
    return value; // value is type cast to type typeName
}
```

Prototyping

```
void cheers(int); // prototype: no return value
double cube(double x); // prototype: returns a double
```

Why Prototype?

- it tells the compiler what type of return value(*double*), if any, the function has,and it tells the **compiler** the number and type of function arguments.

```
double volume = cube(side);
```

- Header file makes C++ compile prototype and function body independently.

What Prototype Do for you? The compiler:

- correctly handle the function return value.
- the correct number of function argument.
- correct type of arguments.

Passing by Value

```
// prototype
double cube(double x);

int main(){
    double side = 5;
    // side = 5 is actual parameter
    double volume = cube(side);
}

// description
double cube(double x){
    // x is formal parameter
    return x*x*x;
}
```

- When function is called, it creates a new type `double` variable called x and initialize x with `x=5`.
- `Actual parameter` is passed to `formal parameter`.
- Function description: `x` is a **private**, **local** variable. After function is terminated, `x` is **destroyed** and memory is freed.
- Multiple formal arguments must be declared separately with demanded types. Even if types are the same.

```
void n_chars(char c, int n);
void test(float a, float b);
```

Functions and Arrays

- ***Array name is address of first element***.

```
arr == &arr[0];
```

- `&` is the address operator. Also, can be used to declare reference.

```
// identical: ONLY in prototype
int sum_arr(int * arr, int n);
int sum_arr(int arr[], int n);
```

- `const`: Prevent arrays from altered

```
void showArray(const double ar[], int n);
```

Function Using Array Range

- Processing arrays in C++ :
    - the kind of data in the array.
    - location of the beginning of the array.
    - the number of elements in the array.
- Another way:
    - specify a `range` of elements.
        - passing the start of elements
        - & end of elements.
- Example:

```
double a[20];
```

```
* two pointers:
    * a : start
    * a+20 : end
```

Pointer and `const`

- a const pointer

```
int age = 39;
const int * pt = &age;
// *pt is const
*pt += 1; // INVALID!!!
```

- pointer-to-a const
    - pointer-to-const must be a const pointer.
    - C++ ***prevent*** you from assigning the address of a `const` to a `non-const pointer`.

```
const float g_earth = 9.80;
const float * pe = &g_earth;

// INVALID!!!!
float * pe = &g_earth;
```

- More comments:

```
int sloth = 3;
const int * ps = &sloth; // (1)
int * const finger = &sloth; // (2)
```

```
* (1): *ps cannot alter the value of sloth. But, *ps can points to another value.
* (2): `finger` only points to `sloth`.
```

Two-dimensional Array

```
// Identical
int sum(int ar[][4], int size);
int sum(int (*ar2)[4], int size);
```

- `[4]` is the columns => # of cols is fixed at 4.

```
int a[100][4];
int b[100][4];

int total1 = sum(a, 100); // sum all of a
int total2 = sum(b, 6); // sum all of b
int total3 = sum(a, 10); // sum first 10 rows of a
int total4 = sum(a+10, 20); // sum next 20 rows of
```

C-Style String

- 3 choices for representing a String:
  - An array of `char`;
  - A quoted `string` constant;
  - A pointer-to-`char` set to the address of a string/
- C-Style `string` has a **terminating character**. Not need to pass the size of the string as an argument.
- Iterate a string:

```
while (*str) {
    // quite when *str == '\0'
    // terminating character
    ...
    str++; // move pointer to the next char
}
```

Adventures in Functions

Inline Functions

- Purpose: speed up programs <- How compiler incorporates them into a program.
- `Normal function`: a program jump back & forth to keep track of different address.
- `Inline function`: compiled code is **inline** with other codes in the program. But, it comes with memory penalty.
- **Common practice**: *omit prototype* and place entire definition where prototype would normally go.

```
// an inline function definition
inline double square(double x) {return x*x;}
```

Reference Variables

- Reference is an ALIAS or alternative name for a previously defined variable.
- Mechanism:
  - works with **original data** instead of with a copy
  - convenient for processing large structures.
  - essential for designing classes.
- Create a Reference Variable
  - `&`: make 2 variables interchangeable.
  - same value
  - same address

```
int rats;
int & rodents = rats;

int main(){
    ...
    rodents++; // both +1
    ...
    // same address
    std::cout << &rats;
    std::cout << &rodents;
    return 0;
}
```

- Reference is declared with initialized value.
- Reference is like a `const` pointer. Cannot refer to another variable.

```
// similar
int & rodents = rats;
int * const pr = &rats;
```

Pass by Reference

```
void swapr(int & a, int & b);
void swapp(int * a, int * b);
void swapv(int a, int b);
```

- `swapr` and `swapp` can swap a and b.
- `swapv` cannot because it just create a copy.

`const` Caveat

- Using const to protect you against programming errors that **inadvertently alter data**.
- Using `const` allows a function to process **both** const and non-const actual arguments.
- Using a `const` reference allows the function to generate and use a **temporary variable** appropriately.

Using References with a Structure

```
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
}

void display(const free_throws & ft);
void set_pc(free_throws & ft);
free_throws & accumulate(free_throws & target, const free_throws & source);
```

- `const` reference cannot be changed.
- **Why return a Reference?**

```
double m = sqrt(16.0); // (1)
cout << sqrt(25.0); // (2)
```

- In (1), value `4.0` is copied to a **temporary location**, then the value in this location is copied to `m`.
- But if returns a reference, things become more efficient.
- `team` is **directly** copied to dup.

```
free_throws & accumulate(free_throws & target, const free_throws & source);

int main() {
    ...
    dup = accumulate(team, five);
}
```

- **BE Carefule!!!!**

  ```
  // wrong !!!
  const free_throws & clone1(free_throws & ft) {
      free_throws newguy;
      newguy = ft;
      return newguy;
  }
  ```

  - `newguy` vanishes as soon as the function terminated.
  - Simple way is to return a reference that was passed as an argument to the function.
  - Second method is to create a new storage.

  ```
  // correct!
  const free_throws & clone2(free_throws & ft){
      free_throws * pt;
      *pt = ft; // copy info
      return *pt;
  }
  ```

  - `pt` points to the structure and `*pt` is the structure.
  - But, always remember to **delete**.

When to Use Reference ?

- Main reasons:
  - Alter a data object in the calling function
  - Speed up a program by passing a reference instead of an entire data object.
- Reference is a different interface for pointer-based code.

Guidelines: Value / Reference / Pointer

- Functions use passed data *without modifying* it:
  - if the data object is **small**, such as built-in data type or a small structure, pass it by value.

    - if the data object is **an array**, use a pointer. Only choice. And, make the pointer const: `const * pt`.
    - if the data object is **good-sized structure**, use a `const` pointer or a `const` reference.
    - if the data object is a class object, pass it by reference.
- Function *modifies* data in the calling function:
    - if the data object is a built-in data type, use a pointer.
    - if the data object is an array, only choice is pointer.
    - if the data object is a structure, use a reference or a pointer.
    - if the data object is a class object, use a reference.

Default Arguments

- Establish a default value using a function prototype.

```
char * left(const char * str, int n=1);
```

- `char *`: return type is a pointer-to-`char`.
- `const char * str`: leave the original string **unaltered**.
- Default arguments are programming-convenience. It greatly reduces the number of constructors, methods, and method overloads needed to define.

```
# include <iostream>
const int ArSize = 80;
char * left(const char * str, int n=1);

int main(){
    using namespace std;
    char sample[ArSize];
    cout << "Enter a string: \n";
    cin.get(sample, ArSize);
    char *ps = left(sample, 4);
    cout << ps << endl;
    delete [] ps; // free old string
    ps = left(sample);
    cout << ps << endl;
    delete [] ps; // free new string
    return 0;
}

char * left(const char * str, int n){
    if (n<0){
        n = 0;
    }
    char * p = new char[n+1];
    int i;
    for (i = 0; i<n && str[i]; i++){
        p[i] = str[i];  // copy charactes
    }
```

```
    while (i <= n){
        p[i++] = '\0'; // set rest of string to '\0'
    }
    return p;
}
```

Function Overloading/Polymorphism

- Functions perform basically the same task but with **different forms** of data.

```
char * left(const char *str, unsigned n);
char * left(const char * str);
```

- But, using single function with default argument is simpler.

Function Templates

Basics

- A *function template* is a generic function description.
- By passing a type as a parameter to a template, the compiler to generate a function for that particular type.
- => ***generic programming***

```
template <typename T>
void swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

- `template <typename T>` and `template <class T>` are interchangeable. But, `typename` is preferred.

```
template <typename T>;
void swap(T &a, T &b);

int main()
{
    ...
    double a=1.5;
    double b=2.6;
    swap(a, b);

    int x=1.5;
```

```
    int y=2.6;
    swap(x, y);
    ...
}
```

- *Function template* does **NOT** make executable programs shorter.
- The above codes generate 2 separate function definitions.
- Typically, templates are placed in a **header file**.

Overload Templates

- Purpose of using templates:
  - Apply the same algorithm to a variety of types.
  - Overloaded templates need distinct function template.

```
// original template
template <typename T>
void swap(T &a, T &b);

// new template
template <typename T>
void swap(T *a, T *b, int n)
```

- NOT all template arguments have to be template parameter types, e.g., `int n`.

Template Limitations

```
template <typename T>
void f(T a, T b)
{...}
```

- The code makes assumptions about what operations are possible for the type.
  - `a=b` is true for type `T` NOT standard built-in type.
  - similarly, `if (a>b)` is not true if they are arrays because the name of array is the address of the first element.

Function Templates Specialization

```
struct job
{
    char name[40];
    double salary;
    int floor;
}
```

- if you want to swap contents in two structures:

```
temp = a;
a = b;
b = temp;
```

- It is OK for you to assign one structure to another with all elements changed.
- But, ***how about only swap some of elements?***

Third-Generalization Specialization

- for a given function name:
    - non template function

```
void swap(job &, job &);
```

- template function

```
template <typename T>
void swap(T &, T &);
```

- explicit specialization template function `template <>`

```
template <> void swap<job>(job &, job &);
```

- specialization overrides regular template.
- non-template function overrides both.

```
// explicit specialization
template<> void swap<job>(job &j1, job &j2);
...

// definition
template<> void swap<job>(job &j1, job &j2){
    double t1;
    int t2;
    // swap salary
    t1 = j1.salary;
    j1.salary = j2.salary;
    j2.salary = t1;
    // swap floor
    t2 = j1.floor;
    j1.floor = j2.floor;
```

```
    j2.floor = t2;
}
```